

AD-A086 811

NAVAL RESEARCH LAB WASHINGTON DC F/6 9/2
FUNCTIONAL DESIGN FOR AN AUTOMATED DATA BASE ERROR DETECTION SY--ETC(U)
MAY 80 6 A WILSON, S B SALAZAR, K L HENINGER
NRL-MR-4223 SBIE-AD-E000 448 NL

UNCLASSIFIED

SBIE-AD-E000 448

NL

101
 201
 301
 401
 501
 601
 701
 801
 901
 1001
 1101
 1201
 1301
 1401
 1501
 1601
 1701
 1801
 1901
 2001
 2101
 2201
 2301
 2401
 2501
 2601
 2701
 2801
 2901
 3001
 3101
 3201
 3301
 3401
 3501
 3601
 3701
 3801
 3901
 4001
 4101
 4201
 4301
 4401
 4501
 4601
 4701
 4801
 4901
 5001
 5101
 5201
 5301
 5401
 5501
 5601
 5701
 5801
 5901
 6001
 6101
 6201
 6301
 6401
 6501
 6601
 6701
 6801
 6901
 7001
 7101
 7201
 7301
 7401
 7501
 7601
 7701
 7801
 7901
 8001
 8101
 8201
 8301
 8401
 8501
 8601
 8701
 8801
 8901
 9001
 9101
 9201
 9301
 9401
 9501
 9601
 9701
 9801
 9901
 10001
 10101
 10201
 10301
 10401
 10501
 10601
 10701
 10801
 10901
 11001
 11101
 11201
 11301
 11401
 11501
 11601
 11701
 11801
 11901
 12001
 12101
 12201
 12301
 12401
 12501
 12601
 12701
 12801
 12901
 13001
 13101
 13201
 13301
 13401
 13501
 13601
 13701
 13801
 13901
 14001
 14101
 14201
 14301
 14401
 14501
 14601
 14701
 14801
 14901
 15001
 15101
 15201
 15301
 15401
 15501
 15601
 15701
 15801
 15901
 16001
 16101
 16201
 16301
 16401
 16501
 16601
 16701
 16801
 16901
 17001
 17101
 17201
 17301
 17401
 17501
 17601
 17701
 17801
 17901
 18001
 18101
 18201
 18301
 18401
 18501
 18601
 18701
 18801
 18901
 19001
 19101
 19201
 19301
 19401
 19501
 19601
 19701
 19801
 19901
 20001
 20101
 20201
 20301
 20401
 20501
 20601
 20701
 20801
 20901
 21001
 21101
 21201
 21301
 21401
 21501
 21601
 21701
 21801
 21901
 22001
 22101
 22201
 22301
 22401
 22501
 22601
 22701
 22801
 22901
 23001
 23101
 23201
 23301
 23401
 23501
 23601
 23701
 23801
 23901
 24001
 24101
 24201
 24301
 24401
 24501
 24601
 24701
 24801
 24901
 25001
 25101
 25201
 25301
 25401
 25501
 25601
 25701
 25801
 25901
 26001
 26101
 26201
 26301
 26401
 26501
 26601
 26701
 26801
 26901
 27001
 27101
 27201
 27301
 27401
 27501
 27601
 27701
 27801
 27901
 28001
 28101
 28201
 28301
 28401
 28501
 28601
 28701
 28801
 28901
 29001
 29101
 29201
 29301
 29401
 29501
 29601
 29701
 29801
 29901
 30001
 30101
 30201
 30301
 30401
 30501
 30601
 30701
 30801
 30901
 31001
 31101
 31201
 31301
 31401
 31501
 31601
 31701
 31801
 31901
 32001
 32101
 32201
 32301
 32401
 32501
 32601
 32701
 32801
 32901
 33001
 33101
 33201
 33301
 33401
 33501
 33601
 33701
 33801
 33901
 34001
 34101
 34201
 34301
 34401
 34501
 34601
 34701
 34801
 34901
 35001
 35101
 35201
 35301
 35401
 35501
 35601
 35701
 35801
 35901
 36001
 36101
 36201
 36301
 36401
 36501
 36601
 36701
 36801
 36901
 37001
 37101
 37201
 37301
 37401
 37501
 37601
 37701
 37801
 37901
 38001
 38101
 38201
 38301
 38401
 38501
 38601
 38701
 38801
 38901
 39001
 39101
 39201
 39301
 39401
 39501
 39601
 39701
 39801
 39901
 40001
 40101
 40201
 40301
 40401
 40501
 40601
 40701
 40801
 40901
 41001
 41101
 41201
 41301
 41401
 41501
 41601
 41701
 41801
 41901
 42001
 4210

FND

DATE _____

FILMED

8-8

DTIC

ADA 086811

NOV 13 1966

PTC

S-1-1-1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Memorandum Report 4223	2. GOVT ACCESSION NO. AD-A086811	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FUNCTIONAL DESIGN FOR AN AUTOMATED DATA BASE ERROR DETECTION SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Final report on an NRL problem.
7. AUTHOR(s) Gerald A. Wilson, Sandra B. Salazar, and Kathryn L. Heninger		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, D.C. 20375		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Research Laboratory Washington, D.C. 20375		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62721N; XF21-244-103; 75-0104-0-0
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE May 15, 1980
		13. NUMBER OF PAGES 56
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Data base management systems Data description language Integrity checking Integrity constraint rules Data base error detection Dynamic transaction		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report documents the design proposed for an automated system for data base error detection. The system, called COPE, will be a front end to the data base management system to catch errors before they enter the data base. It will check for errors in each user's updates and prevent errors due to undesirable interactions between users. It should be invisible to the users during normal processing; when an error is detected, it will operate interactively to inform the user and to request correction.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

1 **UNCLASSIFIED**
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Em

CONTENTS

1. INTRODUCTION	1-1
2. DESIGN GOALS AND BACKGROUND	2-1
3. OVERVIEW OF COPE	3-1
4. CONCEPTUAL FRAMEWORK	4-1
5. KNOWLEDGE APPLICATION SPECIALISTS — THE TASKS	5-1
6. SEARCH STRATEGIST	6-1
7. INTEGRITY CHECKING SCENARIOS	7-1
8. REFERENCES	8-1
APPENDIX A — Internal Conceptual Model Description	
Language (ICMDL) Syntax	A-1
APPENDIX B — Internal Conceptual Model Description	
Language (ICMDL) Semantics	B-1
APPENDIX C — Algorithms Associated with the Tasks	C-1

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DDC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist.	Avail and/or special
A	

DTIC
ELECTE
S JUL 18 1980 D
D

FUNCTIONAL DESIGN FOR AN AUTOMATED DATA BASE ERROR DETECTION SYSTEM

1. INTRODUCTION

In recent years, data base management and management information systems have grown dramatically in size, power, and number. The automation of data collections by business, industry, and government has proceeded with the same fervor as the automation of accounting systems during the late 1960's. People are rediscovering the truth of the old adage "garbage in, garbage out". Although many of the errors in data bases can be caught and corrected by manual checking and correcting procedures, the cost is very high, and the work very tedious. It makes little sense to automate the data storage and management, while relegating the painstaking tasks of error checking and correcting to humans. Therefore, the research and development communities have begun to address the problems of automated error checking and correcting. One of these efforts is the Cooperative Overt Passive Error-Detection (COPE) system which is in the design stage of development at the Naval Research Laboratory.

Virtually all data base management systems (DBMS) incorporate some automated error checking facilities. Typical checks are for proper data format (integer, real, alphabetic, etc.), proper numeric sign, the correct number of data items, and the presence or absence of data in certain fields. While these checks are important and help to prevent some errors, major errors are still found in data bases, indicating that these checks are not sufficient. The purpose of the COPE system is to provide a far more powerful means for error checking. COPE is designed to operate as a stand-alone system working in cooperation with an existing, unmodified DBMS. COPE can be employed both to detect errors in an entire existing data base, which we term static checking, and to check proposed data base updates, which we term dynamic checking.

The COPE system acts as a filter between the users of the DBMS and the DBMS itself, in the sense that it allows the passing of only the valid data base updates. COPE is a passive filter because it is invisible to the DBMS users until a potential error has been detected. When an error is detected the system becomes both overt and cooperative to inform the user of the nature of the error and to aid in correction.

This report describes the design of the COPE system. The design goals and background for this research are presented in section two. Section three gives an overview of the system. Sections four through six describe the three major sections of COPE: the knowledge representation model, the mechanism for applying knowledge, and the heuristic optimizer. A collection of scenarios illustrating the use of COPE appear in section seven.

Manuscript submitted March 13, 1980.

2. DESIGN GOALS AND BACKGROUND

Data base errors can be divided into four types (Eswaran and Chamberlin[14]): security, reliability, consistency, and integrity. Security errors are unauthorized accesses to the data base. Reliability errors are due to DBMS hardware or software malfunctions. Consistency errors are due either to multiple users sharing a single data base or to multiple users sharing more than one copy of a data base. A data base can become inconsistent when multiple updates are processed out of sequence, or when the data base changes in the middle of a user update. Integrity errors include all other errors introduced by active use of the data base system. These may be typing or spelling errors, transmission errors which cause the data to be garbled or transformed between the original source and the data base system, or user errors caused by misunderstandings of the nature or content of the data base. The COPE system focuses on the detection of integrity errors, although its methods also aid in the detection of some consistency errors.

COPE is intended to provide state-of-the-art error detection for Navy command and control applications. Therefore the following design goals have been established for COPE: it should be independent of the applications programs, modifiable and extensible, portable, and logically and physically independent of the data base management system.

Independence from Applications Programs

COPE performs all integrity checking independently of the applications programs in order to insulate each user from the errors of other users. If the responsibility for maintaining the integrity of the data base resides with the user-written applications programs, then no one is really protected. Even if the error checking procedures were provided to the users, the level of protection would be inadequate if the responsibility for employing the appropriate procedures remained with the user.

Modifiable and Extensible

Although most of the techniques employed by COPE have been successfully utilized in other contexts, their effectiveness in combination in a command and control context is not a foregone conclusion. Therefore it is essential that the system be capable of easy modification as the experimentation with COPE leads to new and better techniques. COPE should be extensible so that more sophisticated error checking can be added as the state-of-the-art advances.

Portability

The integrity checking system must be executable on all of the members of a compatible family of computers ranging from minicomputers through complex multi-processing systems. The efficiency and speed of the system would of course be influenced by the choice of machine. The power and complexity of the integrity checking mechanism must be adjustable to the nature of the configuration without requiring reprogramming. The intent is to enable compatible versions of the integrity checker to be employed in the fleet as well as in command centers without requiring a separate development for each application.

Logical Independence from DBMS

In a command control system it is essential that the system be able to function even when the load becomes heavy as in a crisis situation. Therefore if the integrity checking system becomes a bottleneck, it must be possible to shut it down dynamically until the bottleneck is alleviated. The DBMS system must not depend on the integrity checker for its own correct operations.

Since integrity checking is as important during a crisis as it is normally, the best approach is to disable only the minimum amount of integrity checking necessary to alleviate the bottleneck. Thus the checking of data types for field compatibility might still be possible during a crisis although more complex checks involving additional data base accesses were disabled.

Since data management systems applicable to the command and control environment have been under development for several years, the integrity checker will have to be retrofitted to the data management system. Retrofitting would be far more cost effective and less traumatic if it can be accomplished without rewriting or significantly modifying the existing data management system.

Physical Independence from DBMS

There are many issues involved in designing COPE to operate on a processor which is physically distinct from the DBMS. The advantages are those advocated for back-end data base management by Canaday et. al.[6]. A system so constructed could execute on a specialized machine, could adapt to different computer systems, and could operate asynchronously with respect to the DBMS. There are also two potential disadvantages to this approach: the inability to share resources and the additional hardware costs. To justify the decision to make COPE physically detachable, these issues will be discussed in more detail below.

The hardware and software requirements for an integrity checker are likely to be quite different from those of a large data management system. Unlike the data management system, the integrity checker need not manipulate a large data base, large volumes of output to users, and potentially conflicting updates and searches. Tailoring the hardware for the integrity checker may result in a more efficient machine for the purpose.

In a large command control system there may be many distinctly different computer systems. The hardware and software at a location responsible only for maintaining and reporting localized information may be quite different from that at the regional facility responsible for many smaller locations. In addition, some locations may have specialized functions which require different types of data management facilities. One would expect to find smaller data bases on an aircraft carrier than at the regional fleet headquarters. Another type of system might be appropriate for the Fleet Numerical Weather Service, which is the central source for weather data for all the oceans of the world which may be of concern to Naval vessels. Yet all of these computer systems have data bases which require integrity checking. A physically distinct integrity checking system should be able to be interfaced with a large variety of computer systems with few required modifications.

It is not always necessary for the integrity checker and the data manager to address the user-presented queries and updates in the same order. The integrity checker might suspend the processing of a difficult item to handle a number of more easily checked items and pass them on to the data base manager. Then while the data base manager completes its processing, the integrity checker can continue with the original item to be checked. The approach is a form of multiprocessing, but with physically and functionally distinct processors and few if any shared peripherals.

If the error detection system is physically distinct from the DBMS, the inability to share resources is a disadvantage. If it is the case that bottlenecks on one processor are matched by idle periods on the other processor, an integrated system might have enabled the idle resources to be used. Additional hardware costs for the independent processor are less likely to be a deterrent. The total dollar amount for the additional hardware may be quite small with the current trends in electronics. Also, the use of an independent hardware and software system will prevent any expenses for redesigning or modifying the existing DBMS which, given the costs of software support, could be considerable. Therefore an independent error detection mechanism like COPE can have advantages in both cost effectiveness and system adaptability which far outweigh the cost of the additional hardware.

Background

A number of researchers have examined the problems of data base integrity. A very complete description of the functions needed in the data base description language to handle integrity checking is given in Graves[19]. Many aspects of the integrity problem were also documented in CODASYL[8], Date[11], Eswaran and Chamberlin[14], and Hammer and McLeod[20]. More theoretical examinations of the problems have been published by Fernandez and Summers[15] and by Floretin[16]. All of these research efforts share the common theme with COPE: perform integrity checking independently of the applications programs.

Unlike COPE, few of the systems described in the open literature have employed the approach of keeping the error detector physically and logically distinct from the data base manager. The developers of the experimental data management system (XDMS) [6] investigated the tradeoffs involved in choosing between an integrated system versus physically distinct processors. The COPE design goals were influenced by their finding that the evidence strongly supports the separate processor approach.

Since COPE is required to execute on computers of various sizes, it is important to consider the data management systems implemented on minicomputers. McLeod and Meldman[25] developed a relational data management system called the Relational Inquiry and Storage System (RISS) for PDP-11 machines under the RT-11 operating system. RISS was interactive but had no capability for integrity checking.

Another system for PDP-11 type computers which is still under active development is the Interactive Graphics and Retrieval System (INGRES)[35]. Like COPE and RISS, this system uses a relational representation for the data

base. Instead of requiring the user to communicate in the language of the DBMS as does COPE, INGRES provides a relatively sophisticated query language called QUEL. Through QUEL a user can pose questions requiring multiple, interrelated DBMS retrievals and each user question is translated by INGRES into the appropriate set of data management system queries. Although some very limited integrity checking on single relations is claimed for INGRES[34], this feature is not yet fully operational. The current version of INGRES also can support only one user at a time on any given data base, although it can support multiple users with multiple independent data bases. This differs from COPE, which is designed to support multiple users of the same data base.

An important issue raised by the INGRES project is whether a complex integrity checker will be inherently slow. INGRES, which has very limited error checking in comparison to COPE, has been noted to be quite slow, which its originators claim to be partially due to the system itself and partially due to the UNIX operating system under which it runs. The question of throughput time is critical in the command and control environment and must be considered throughout the design and development of COPE.

Another minicomputer data management system under initial development is the Command Center Information System (CCIS) [33] now being designed at NOSC. Like COPE, this system is aimed specifically at the command and control application. However, CCIS has no integrity checking capabilities.

3. OVERVIEW OF COPE

Introduction

This section provides an overview of COPE. It discusses COPE's place in the DBMS environment, its internal and user languages, the underlying data base model, the types of error checking, and the component breakdown. Finally, several design problems are presented and resolved.

Testing environment and methods

The examples used in this paper are taken from the Navy command and control problem area, which is the initial target application of the COPE system. The environment in which COPE is to be tested is that of the Advanced Command Control Architectural Testbed (ACCAT), which is described in reports from the Naval Ocean Systems Center (NOSC) [3,4,5,28]. ACCAT is a secure subnet of the ARPANET. The ACCAT database is in a relational format and was created from a snapshot of an operational Navy data base, NWSS. A smaller, unclassified version of the ACCAT data base, called the BLUEFILE data base [5] is available on the open ARPANET. The ACCAT data base is supported by a DBMS called the Datacomputer [10], which uses a DBMS language called Datalanguage.

Several approaches, varying in ease of implementation and validity of results, can be taken in testing COPE. The simplest test is to use human generated update streams which would include errors. This would be easy to implement but may provide invalid errors due to the artificial operating conditions. A second alternative is to use an automated update mechanism which would generate random semantic errors. A considerable development effort would be needed to create a mechanism which would generate errors requiring the use of the more complex rules. Another approach would be to use data base updates obtained from an on-line log of real messages intended for the NWSS data base. While this approach would be the most realistic, there would be problems in tapping into the operational environment, cleansing classified data, and finding the errors. The choice of technique will depend on the resources available at the time of testing and evaluation.

The results of the analysis should provide a comprehensive view of the power and utility of the COPE system within the ACCAT domain. In addition, by measuring the cost of execution of COPE as the user inputs are processed, the test should also provide a guide to the actual cost of utilizing a system like COPE in an operational environment.

Relation to DBMS and Users

COPE functions as an invisible front-end to the DBMS, invisible both to the DBMS and to the user. Since the DBMS cannot distinguish between COPE and any other user, COPE requires no special interface and no special modes from the DBMS. COPE expects the DBMS to behave exactly as it does with any other user. From the user's point of view, COPE is a natural extension of the normal DBMS. When a potential error is found, the user interacts with COPE either in a binary answer mode (yes/no responses to COPE questions) or in the language of the DBMS. COPE makes no assumptions about the size of the DBMS data base or the power of the processor on which COPE is to operate. It is a major design goal that different size instances of COPE should look and behave identically from the viewpoint of both the users and the DBMS.

In order to maintain this independence from the DBMS, COPE must be able to handle a wide variety of error checks, even if some are duplicates of checks performed by the DBMS. If COPE were to be prohibited from making the simple quick checks such as format errors, it might waste resources on complex checks which were unnecessary. Duplication of DBMS error checks is an acceptable penalty for being able to interface to an existing DBMS without modifying it and being able to attach COPE to different DBMS's.

Languages

Several languages are employed by COPE for internal processing and communications with the outside world.

To set up a new version of COPE, a human referred to as the data base expert must supply the knowledge required for error detection. A high-level, easy-to-use, powerful language will be needed to support this function in production versions of COPE. This language will not be developed in the first phase of the COPE project since the issues involved are primarily those of human engineering and are peripheral to the primary purpose of evaluating the usefulness of the COPE approach.

During error-free operation, the user and the DBMS appear to be linked directly because COPE communicates with both in the language of the DBMS. However, to achieve independence from any particular DBMS, the language internal to COPE is a special language called the Internal Conceptual Model Description Language (ICMDL). A translator will employ standard table driven parsing techniques to translate DBMS language statements into ICMDL and vice versa.

ICMDL is a declarative language based on first order predicate logic and set theory. It incorporates many of the characteristics of production rules [12], partitioned semantic networks [21], and extended pi-clauses [36]. ICMDL is similar in its expressive power to the languages ALPHA [9] and QUEL [2], which are based on the relational calculus. Unlike these two languages, ICMDL is declarative rather than procedural and sequential. There are two reasons for this approach. First, a sequential language can force unnecessary arbitrary choices to be made in the expression of a rule. It is more natural to express the rule in the same sequence-independent and algorithm-independent manner in which it is stated in English. Second, a declarative language makes the specification of the rules independent of the processing algorithms of both COPE and the DBMS, enabling COPE to apply rules to the data base in parallel.

Appendix A contains a formal BNF description of the syntax of ICMDL and Appendix B contains an informal description of the associated semantics of the language. ICMDL is not intended as a user language; it is strictly the internal representation language for the conceptual framework. Therefore, every choice between readability and efficient machine use was made in favor of the machine. Because we do not yet know the best method for describing any particular error constraint, the emphasis on ICMDL was to provide as much generality as possible in expressive power. Once the COPE system is actually built, part of the testing and evaluation will be to determine the utility of the various alternatives to expressing different types of constraints.

Underlying Data Base Model

COPE views the DBMS data base as a relational data base, regardless of its actual representation. A relation is a set of n-tuples (Date [11]). Each n-tuple consists of n arguments, each of which can only take on values from a specified domain. The value which an argument takes on is said to instantiate the argument. Since COPE error checking facilities operate on collections of tuples, COPE maps the actual data base structure into a relational structure.

The format of a COPE tuple is: (relation arg_1: value ... arg_n: value). For example, the statement, "The captain of the Enterprise is Perry", is equivalent to the tuple: (ship name: Enterprise captain: Perry). The relation in this tuple is "ship" and there are two arguments, "name" and "captain", which have the values "Enterprise" and "Perry", respectively.

Types of Error Checking

The types of error checking performed by COPE can be categorized according to the operations on tuples. There are two basic categories: checking for the proper construction of a single tuple, and checking that a particular correspondence holds between two or more tuples. Figures 3-1 and 3-2 give examples of errors detected by each of the checking methods described in this section.

The simplest checks applied to a single tuple are those which apply to single arguments. Syntactic checks such as format (line 1), range (line 2), and valid values (line 3) take place during translation of the user update from the DBMS language to ICMDL. Checks for spelling (line 4) and class membership (line 5) involve the semantic content of the data. These checks assure that the item instantiating an argument belongs to a predefined semantic class.

The more complicated error checks applied to a single tuple verify that the data agrees with semantic conditions expressed as rules, that is, that certain conditions hold between the data in two or more argument positions of the same tuple (line 6).

After checking each tuple individually, COPE applies tests which involve more than one tuple. Some tests check for consistency among several tuples with the same relation (line 7), while others check that certain conditions hold for tuples with different relations (line 8).

Error checking which involves inter-related conditions and complex calculations (line 9) is beyond the scope of this phase of the COPE project. Such checking involves breaking down a complex rule into a collection of simpler rules and employing an automatic inference mechanism to direct the interaction of the simpler rules. Approaches like this have been employed in systems such as QA3.6[30], MRPPS 3.0[26,27,36], MYCIN[32], and PROSPECTOR[13]. The COPE design incorporates a foundation for building complex checking mechanisms.

Line no.	Example (constraint rule, English statement, error explanation)
1	<p>Rule: Syntax check (format)</p> <p>Input: The length of the Kennedy is 1A5 meters.</p> <p>Error: illegal format - 1A5 is not an integer</p>
2	<p>Rule: Syntax check (range)</p> <p>Input: The Kennedy's speed is 85 knots.</p> <p>Error: illegal range - 85 is not within the bounds specified for ship speed</p>
3	<p>Rule: Syntax check (unit definition)</p> <p>Input: The Sea Wolf's weight is 350 knots.</p> <p>Error: improper unit definition - weight is measured in tons, not knots</p>
4	<p>Rule: Spelling</p> <p>Input: The Enterpris is in Philadelphia harbor.</p> <p>Error: "Enterpris" is not in the set of legal ship names, but "Enterprise" is included</p>
5	<p>Rule: The captain of a ship must have rank of at least captain.</p> <p>Input: Seaman Wolf is captain of the Enterprise.</p> <p>Error: Wolf cannot be captain because "seaman" is a rank lower than "captain"</p>

Figure 3-1: Examples of Errors Handled by COPE - Simple Errors

Line no.	Example (constraint rule, sample tuple(s), error explanation)
6	<p>Rule: The destination city must be in the destination country.</p> <p>Sample tuple: (convoy id: American, dest_city: Havana, dest_country: Brazil)</p> <p>Error: For a tuple with relation "ship", the values of the arguments "dest_city" and "dest_country" must be consistent.</p>
7	<p>Rule: All of the ships in a particular convoy must be headed for the same country.</p> <p>Sample tuples: (ship name: Enterprise, convoy: ATL, dest_country: Cuba) (ship name: Kennedy, convoy: ATL, dest_country: Cuba) (ship name: Constellation, convoy: ATL, dest_country: Brazil)</p> <p>Error: All tuples with the relation "ship" which have the same value for the argument "convoy" must have the same value for the argument "dest_country".</p>
8	<p>Rule: A ship's speed may neither be unreasonably slow nor beyond its capabilities.</p> <p>Sample tuples: (ship name: Kennedy, norm_speed: 6 knots, max_speed: 16 knots) (trackhist name: Kennedy, speed: 17 knots)</p> <p>Error: All tuples with relation "trackhist" that have a certain value for the argument "name" must be consistent with the tuple with relation "ship" that has the same value for the argument "name". Specifically, the value of the argument "speed" of the "trackhist" relation must be within the bounds indicated by the "norm_speed" and "max_speed" arguments of the "ship" relation.</p>
9	<p>Rule: The speed and course of a ship is related to its destination, the speeds and courses of the ships traveling with it, the weather conditions, land masses in its straight line path, and its fuel requirements.</p> <p>This type of rule is too complex to be handled in the current phase of COPE.</p>

Figure 3-2: Examples of Errors Handled by COPE - Complex Errors

Components

The COPE system consists of three components: the conceptual framework, the tasks, and the search strategist. The conceptual framework is the form used for representing the knowledge base of the error-detection system. It includes all the information needed to communicate with the users and the DBMS, and to perform the error detection. The tasks apply the knowledge of the conceptual framework to the data base and the user inputs in order to detect errors. The search strategist is the heuristic optimizer which monitors the system load and controls the execution of the tasks. The search strategist determines which tasks run and what rules they apply to an update. The tasks receive user inputs, apply the rules of the conceptual framework, and query the DBMS.

Design Problems

Two design problems affect the structure of COPE: grouping updates into transactions and avoiding concurrency problems.

It may require more than one update to bring a data base from a state in which all the constraint rules are satisfied, which is called a correct state, into another updated correct state. For example, suppose we have a data base about ships with a constraint rule requiring each ship to have exactly one captain. When the captain of a ship is being changed, there is no way to reassign the old captain and assign the new captain without momentarily violating the constraint rule, unless the updates happen simultaneously. A collection of updates that must occur simultaneously in order for an integrity constraint to hold always is called a transaction by Zloof[15].

There are several approaches to grouping updates into transactions for error detection purposes.

- 1) The user defines a transaction by inserting a special 'end-of-transaction' symbol after each collection of related updates. This has the advantage of being flexible, but the disadvantage of being very dependent on the user understanding both the transaction concept and the collection of constraint rules. In addition it may be very difficult to help the user determine why a given transaction failed to satisfy all the constraint rules.
- 2) The transactions are pre-defined within the error checking system. This greatly eases the burden on the user, but requires that the error detection system have a very large set of pre-defined transactions, and that the data base expert be prepared to change the transaction set whenever the constraint rule set is changed.
- 3) Transactions are defined dynamically by means of the constraint rules. Each update is associated with a collection of rules that reference one or more relations, forming a relation set. An input is considered part of the current transaction if its relation set has members in common with the transaction's relation set. When an input is received that does not have a matching relation, the current transaction is suspended and tested. The updates are passed on to the DBMS only if they produce a correct state. If they do not, a potential error has been detected.

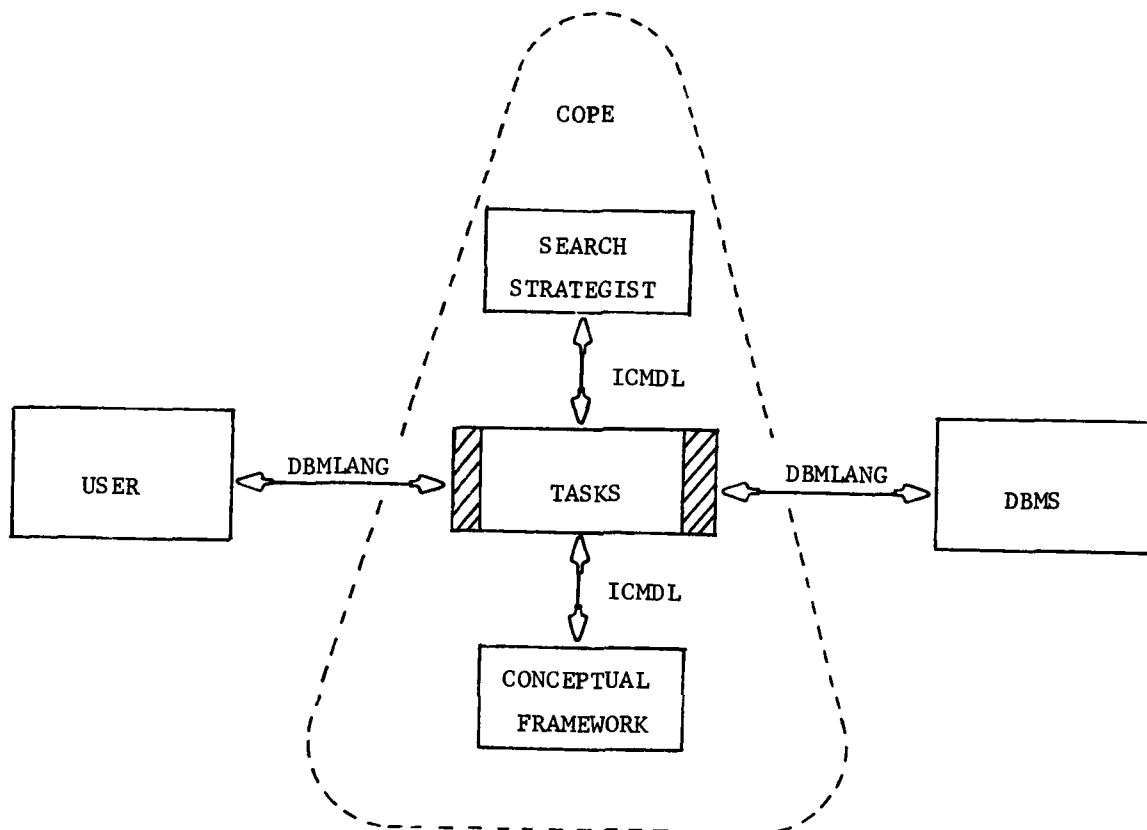






Figure 3-3: An Overview of COPE

Legend

-  COPE programs invisible to both user and DBMS
-  translator programs
-  modules
-  communications paths
- DBMLANG data base management system language

The third alternative has been chosen for COPE. An incomplete transaction is called a user context in COPE. The context represents a temporarily altered version of the data base that is available to no one but that user.

The error checking procedure is in danger of causing concurrency problems because updates can be performed in a different order than they are entered. For example, delaying updates to form a complete transaction can re-order the handling of the updates. Therefore, whenever an update is being checked, the checking task must always employ the most recent copy of the data base. Interaction between different user contexts must be controlled because these contain only hypothetical updates which are not guaranteed to satisfy the constraint rules. This implies that there must be careful controls over the feedback of updating information into user contexts which have not been completed.

These two design considerations can be satisfied if each user appears to be running with an 'independent' copy of COPE, and all interactions between user contexts are well defined and handled by a global supervisor, called the search strategist. Each user is provided with a virtual copy of each program unit, except for those portions of the system where interactions between contexts must occur, in which case a single global program unit must take control.

With this approach, the space required to handle any given user can be made dependent primarily on the complexity of the constraint rules and the number of updates which must be accumulated in the context before a correct state has again been reached. Therefore, by controlling the nature of the constraint rules, the size and power of the hardware required to execute the tasks of COPE can be reduced. The design facilitates the extension of COPE to a truly physically distributed system, because the virtual copy associated with one user can be executed on a processor distinct from that of another user without any increase in the communication between them.

4. CONCEPTUAL FRAMEWORK

4.1 Overview of the Conceptual Framework

The conceptual framework is the skeleton which is to be filled in with the knowledge base of COPE. While the framework is the same for all COPE systems, the knowledge depends on the specific data base and therefore differs for each system. A human data base expert must initially supply COPE with the knowledge in the specified format (in the experimental version, it must be formatted directly in ICMDL); the knowledge base then does not change during execution of the system.

The framework consists of four components:

- (1) the data base structure description
- (2) the relation templates
- (3) the semantic dictionary
- (4) the constraint rules

The data base structure description specifies the syntactic structure of the target DBMS data base. The relation templates describe the data base according to the relational model. The semantic dictionary lists the semantic classes for each constant in the data base. The constraint rules are conditions that must be true for the target data base to be free of errors.

Examples of the components of the conceptual framework are given in Figures 4-1 through 4-6. These examples are referenced in the following sections.

4.2 The Data Base Structure Description

In order to communicate with the DBMS and to perform syntax checks, COPE must have a syntactic representation of the data base. To set up a version of COPE for a particular data base, the data base expert constructs the DBSD by converting the structural description of the data base from the DBMS language into the file format of ICMDL. This part of ICMDL is intended to be very close to the language used for any particular DBMS, so that the effort required for the data base expert to translate from one to the other is minimized. It is also intended to be sufficiently general so that it can be mapped to a large class of DBMS languages. Figure 4-1 illustrates a description of a Navy data base in Datalanguage and the corresponding description in ICMDL. The Datalanguage-ICMDL mapping is straight-forward. For example, a "file" in Datalanguage is a "FILE" in ICMDL and a "structure" in Datalanguage is a "RECORD" in ICMDL.

DATALANGUAGE DESCRIPTION:

```
create bluefile.ship file list, p=eof, chapter
  ship_struct structure
    name string (,20)
    captain string (,20)
    readings_count integer
    readings_list (,10,100), c=readings_count
      stats structure
        time structure
          hour integer
          minute integer
        end
        heading integer
      end
    end
  end;
```

ICMDL DESCRIPTION:

DATA BASE STRUCTURE DESCRIPTION:

```
(FILE ship (LIST WITH TERMINATOR = EOF OF
  (TYPE ship_struct (RECORD
    ((name: (TYPE name (ARRAY (1 20) OF CHAR)))
    (captain: name)
    (readings_count: integer)
    (readings: (LIST WITH SIZE = readings_count OF
      (TYPE stats (RECORD
        ((time: (TYPE time (RECORD
          ((hour: integer)
          (minute: integer))))
        (heading: integer))))))))))
```

Figure 4-1: Description of a Data Base in a DBMS Language and the Corresponding Data Base Structure Description in ICMDL

During the execution of COPE, the DBSD provides the tables used by the translator to convert user inputs from the particular DBMS language into ICMDL, to convert COPE queries addressed to the DBMS to the DBMS language, and to perform all syntactic checking of user updates, such as type checking and detection of non-existent file or field names.

Since COPE is intended to work with any DBMS, ICMDL must be sufficiently rich to enable any structure to be described in the DBSD. The approach used in ICMDL is to incorporate and extend the declaration facilities which are included in the PASCAL programming language definition [31]. This permits the definition of arrays, lists, sets, bags (unordered lists), and records (complex structures). Because any new data structure type can be built out of collections of other data structures, definitions of complex structures can be constructed easily.

4.3 Templates for the COPE Relations

The COPE relation templates describe a particular data base according to the relational model in order to facilitate the application of semantic error checks. The relation templates, set up by the data base expert, define the COPE relations and the mapping of each argument of each relation to the corresponding field or structure in the DBSD. Additionally, they specify which rules are to be applied to each argument.

Figure 4-2 gives an example of the relation template which corresponds to a segment of the DBSD. In order to form the mapping between the DBSD file and the corresponding COPE relation, the COPE design requires that the names of the file and the relation be the same. A relation template consists of argument declarations, one for each argument of the relation. Each declaration has three parts: the relational argument name, the DBSD path name, and the rule list. The relational argument name is the name used within COPE to reference the argument. The DBSD path name is a unique name formed as the concatenation of the names of all the parent structures up through the file name. It is used by the translator for referencing the corresponding field in the DBSD. The rule list specifies the names of the rules which should be checked when that argument is to be changed.

The data base segment illustrated in Figure 4-2 is a file of records about Navy ships, where each record contains the name of the ship, the name of the ship's captain, and a variable length list of the recent position history of the ship. The first argument declaration specifies that the relational argument "s_name" corresponds to the "name" portion of the "ship_struct" portion of each element of the file "ship". The last part of the argument declaration indicates that the rule "A1" is associated with the argument "s_name". Figure 4-3 shows several tuples which are instantiations of the "ship" relation template.

ICMDL DESCRIPTION:

DATA BASE STRUCTURE DESCRIPTION:

```
(FILE ship (LIST WITH TERMINATOR = EOF OF
  (TYPE ship_struct (RECORD
    ((name: (TYPE name (ARRAY (1 20) OF CHAR)))
    (captain: name)
    (readings_count: integer)
    (readings: (LIST WITH SIZE = readings_count OF
      (TYPE stats (RECORD
        ((time: (TYPE time (RECORD
          ((hour: integer)
          (minute: integer))))
        (heading: integer))))))))))
```

COPE RELATION TEMPLATE:

```
(RELATION ship TEMPLATE
  ((s_name:: ship_struct.name: (A1))
  (s_captain:: ship_struct.captain: (A1 I1))
  (t_stats:: (LIST OF (RECORD
    ((s_time:: stats.time: (A1 I1 I1))
    (s_heading:: stats.heading: (A1))))))
```

Figure 4-2: Example of a Data Base Structure Description and the Corresponding Relation Template

```
(ship s_name: Constellation s_captain: Jones t_stats: (s_time: 1300
  s_heading: 40NW20))
(ship s_name: Kennedy s_captain: Smith t_stats: FREE))
```

Figure 4-3: Example of Data Base Tuples which Use the "ship" Relation

4.4 Semantic Dictionary

Each argument position of a tuple is semantically restricted to a certain group of items. For example, in tuples with the "ship" relation (see Figure 4-3) the argument "s_name" must be filled by the name of a ship and the argument "s_captain" must be filled by the name of a person who has rank of at least "captain". Each group so defined, called a semantic class, is a collection of items which share a common property or meaningful relationship. In the example, the legal values for the first argument must be drawn from the class "shipname" and those for the second argument from the class "scaptain". Thus the legal entries for an argument position can be specified in a rule implicitly by naming the semantic class, rather than explicitly by naming all the possible values.

In the process of grouping items into semantic classes, it becomes obvious that some classes are subclasses of other classes. For example, the class "scaptain" (names of persons who are captains) is a subclass of the class "pname" (names of persons). It is possible to form a network or hierarchy of the classes, with the bottom level consisting of the constants of the data base. This hierarchy of semantic classes is called an ISA-hierarchy or semantic network [29]. For example, Figure 4-4 is a graphic portrayal of the hierarchy, which shows the relationships between semantic classes ("scaptain", "pname", etc.) and constants ("Smith", "admiral", etc.). To determine if a constant is a member of a given semantic class, one need only determine if there exists a path between that constant and the semantic class in the ISA-hierarchy. For example, "Smith" is a member of "pname" since "Smith" is a member of "scaptain" and "scaptain" is a member of "pname".

The component of the conceptual framework which defines the semantic classes and semantic network is called the semantic dictionary. It is represented in COPE as a collection of statements, entered by the data base expert, which specify the classes and subclasses to which each argument of each relation belongs. The format of these statements (commonly called ISA statements) is:

(ISA semantic_class_name class_derivation)

The semantic class name is an identifier by which the class is referenced in the rules. The class derivation specifies how the class (set, bag, or list) is formed from the relations in the data base, using relational operators (Date [11]). For example, Figure 4-5 shows the ISA statements corresponding to the network of Figure 4-4. The class "prank" is a list, which is explicitly defined. The ordering of the list is from lowest to highest, so that "seaman" is less than "ensign" ... is less than "admiral". The class "scaptain" (names of ship captains) is a set, which is formed by taking each value of the argument "p_name" (names of persons) from all the tuples of the "person" relation which also have "captain" as the value of the argument "p_rank" (persons with rank of captain).

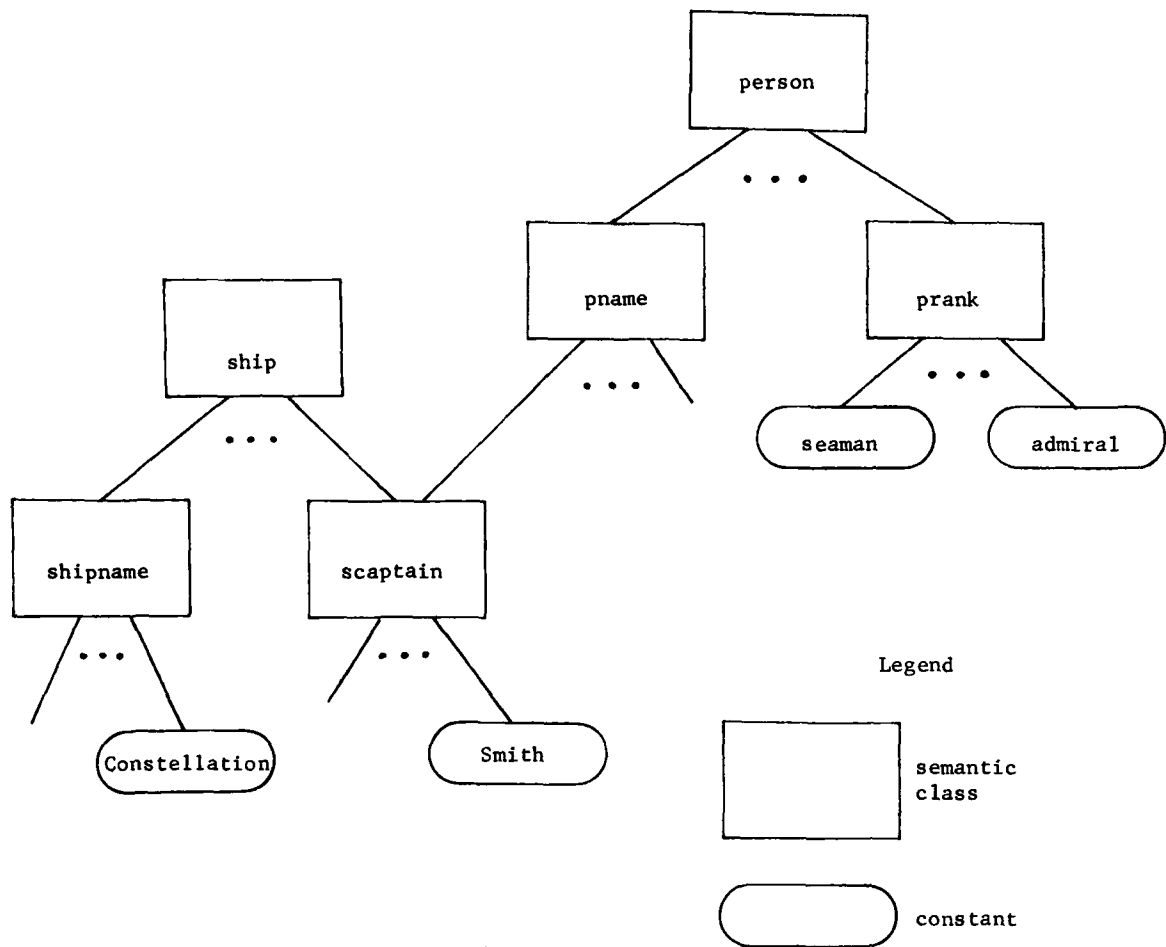


Figure 4-4: A Portion of the Semantic Dictionary,
Graphic Portrayal of the ISA Hierarchy

```

(ISA shipname (SET := (PROJECT (ship (:= s_name)))))
(ISA scaptain (SET := (PROJECT (person (:= p_name) (p_rank = 'captain')))))
(ISA pname (SET := (PROJECT (person (:= p_name)))))
(ISA prank (LIST := (seaman ensign captain general admiral)))

```

Figure 4-5: A Portion of the Semantic Dictionary,
ICMDL Definition of ISA Hierarchy

4.5 Constraint Rules

The COPE constraint rules are semantic restrictions on the data base which must be true at all times. They are represented in ICMDL as predicate-calculus-like statements and are initially entered into the system by a data base expert.

When a user submits an update, only those rules associated with the affected portion of the data base need to be applied. In order to facilitate the selection of the appropriate rules, the rules are associated with the relation templates. Each argument declaration of the template includes a list of rules that should be checked when that argument is updated. Thus, a complete list of rules needed to verify an update can be quickly assembled by referencing the declarations for all the affected arguments.

An ICMDL rule statement consists of six parts. The name is the identifier by which the rule is referenced in the templates. The pattern lists the variables used in the rule, which are instantiated to fit each particular update. The formula is the predicate-calculus expression of the semantic constraint. The action is a list of functions which specify COPE's response to failure of the rule. The cost is a number which indicates the effort involved in applying a rule. The payoff is a number which indicates the likelihood that an error will be detected by applying this rule. Cost and payoff will be discussed in the section on the search strategist.

Some examples of the name, pattern, and formula portions of the constraint rules referenced in the template of Figure 4-2 are given in Figure 4-6. The rules presented here belong to the category of constraint rules which check individual tuples. Rule A1 checks the semantic class membership of the items in each argument position of a tuple with the "ship" relation. Rule I1 tests that a ship has exactly one captain and therefore must check that appropriate conditions hold across two argument positions of the tuple. Rule I11 checks for a valid value of a particular argument, "stats", in a tuple involving the "ship" relation.

The pattern consists of the relation name followed by an argument term for each argument of the relation. The argument term either associates the argument with a variable or declares the argument a free variable. For example, the pattern for rule I1 in Figure 4-6 states that the value supplied for argument "s_name" is associated with the variable "x", the value for "s_captain" is associated with the variable "y", and "t_stats" is a free variable which is not involved in the rule.


```

(RULE
name:    A1

pattern: (ship $s_name: (:= s_name x1) $s_captain: (:= s_captain x2)
          $t_stats: ($s_time: (:= s_time x3) $s_heading: (:= s_heading
          x4)))

formula: (ship $s_name: (s_name = shipname) $s_captain: (s_captain = scaptain)
          $t_stats: ($s_time: (s_time = stime) $s_heading: (s_heading =
          sheading)))

(RULE
name:    I1

pattern: (ship $s_name: (:= s_name x) $s_captain: (:= s_captain y)
          $t_stats: FREE)

formula: (((TYPE z (SET OF CHARACTER))
          (AND (FOREACH x) (EQUAL (COUNT (SET_UNION y z)) 1)
          (ship $s_name: (s_name = x) $s_captain: (:= s_captain z)
          $t_stats: FREE))))))

(RULE
name:    I11

pattern: (ship $s_name: FREE $s_captain: FREE
          $t_stats: (:= $t_stats y))

formula: (CHECKTIME y))

```

Figure 4-6: Some Constraint Rules of the Example Conceptual Model

The formula specifies one or more conditions which must be satisfied. These conditions are stated as first order predicate calculus statements which may utilize the semantic classes, sets, bags, and lists, built-in special purpose predicates, and data base accesses. For example, the formula for rule I1 declares a local variable "z" with the statement: (TYPE z (SET OF CHARACTER)). The variable "z" is to be the set of captains of the ship with name equal to the variable "x" which is supplied in the user input. A data base access will be required to determine this set. The size of the set formed by taking the union of the set "z" and the set "y" of supplied captains' names must be 1 (ie., "z" must be the null set and "y" must have one element or "z" and "y" each have one and the same element). The formula portion of rule I11 consists solely of a call to the special purpose predicate "CHECKTIME". The formula of rule A1 tests that each argument is a member of the specified semantic class. The format of this type of formula is:

(relation \$arg_x1: (arg_x1 = semantic_class_name) ...)

where "=" is read as "is a member of".

If a rule fails to be satisfied when it is invoked, it is necessary to perform some action to inform the user that an error may have been detected. The particular action or actions to be performed may be specified by the action portion of the constraint rule. The types of actions may be as simple as informing the user of the nature of the error and rejecting the data, or as complex as permitting the user to investigate other aspects of the error and the nature of the constraint rules themselves in order to make the necessary changes which would make the data base consistent.

5. COPE'S KNOWLEDGE APPLICATION SPECIALISTS -- THE TASKS

5.1 Overview of the Tasks

The tasks apply the knowledge of the conceptual framework to the data base and to the user inputs in order to detect errors. The tasks intercept each user input (update or query), decide which rules are relevant, apply the rules, and either pass the verified user input on to the DBMS, or inform the user of an error.

For each user there are five tasks which may be employed during error detection. Table 5-1 gives the name of each task and a brief list of its functions. Figure 5-1 depicts the information flow through COPE and the relationships between the tasks. Supplementing this discussion of the tasks is Appendix C which contains detailed algorithms.

PROGRAM UNIT	FUNCTIONS	PARTS OF CONCEPTUAL MODEL ACCESSED
SETUP	Normal user interface DBMS Language to ICMDL translation for user updates Syntactic error checking Context determination Constraint rule selection	Relation templates DBSD Constraint rules
VERIFY	Constraint rule application	Constraint rules
LOCAL_DATA	Interface to DBMS ICMDL to DBMS language translation for queries directed to DBMS Control of local copies of data	Constraint rules DBSD
BACKTRACK	Adjust partially verified rule when data base or user inputs change	Constraint rules
MESSAGE/ RESPONSE	User interface for error correcting dialogue with user DBMS language to ICMDL translation for user corrections	Constraint rules DBSD

Table 5-1: The COPE Program Units for the Tasks

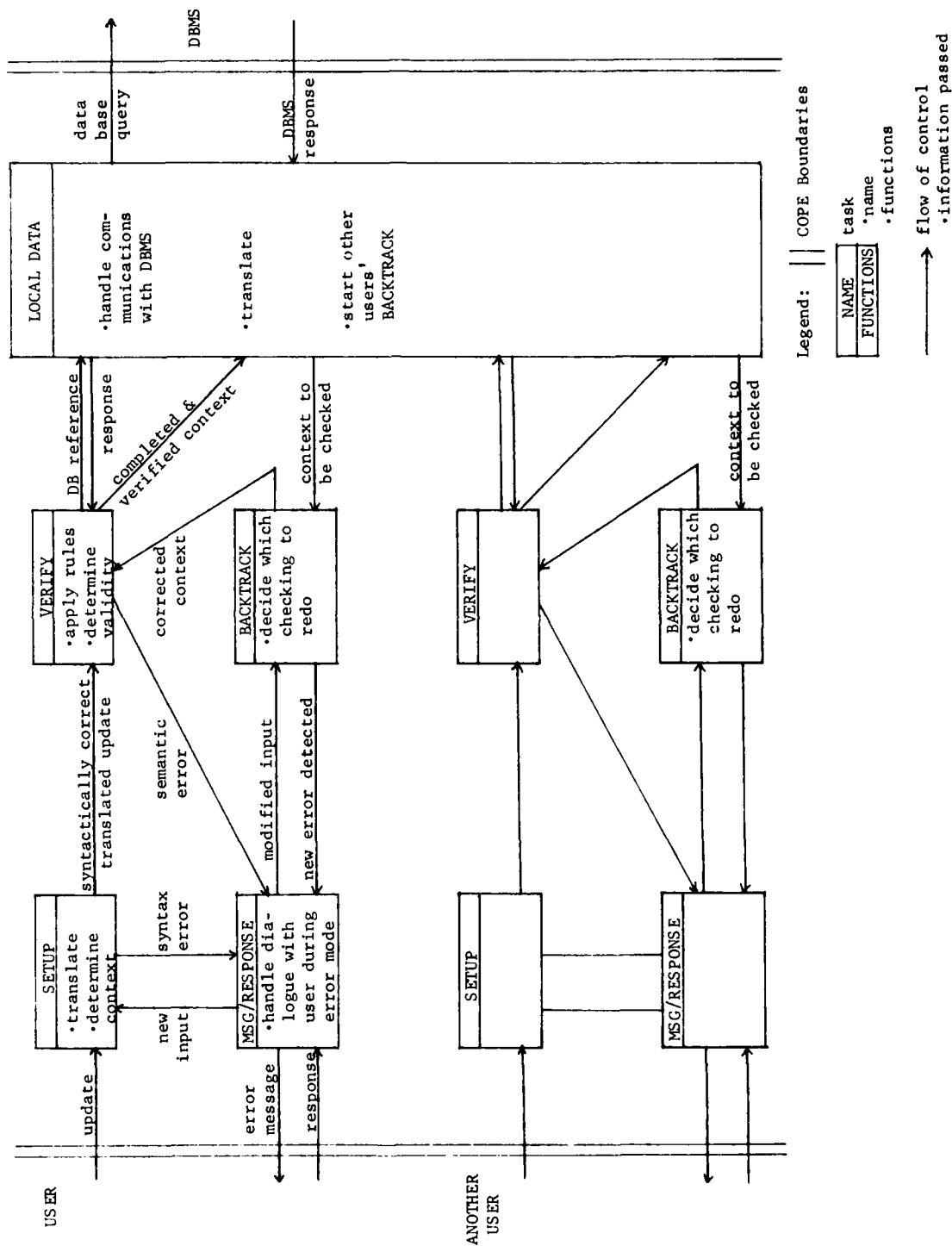


Figure 5-1: Information Flow Through COPE

The SETUP task and the MESSAGE/RESPONSE (M/R) task comprise the user-COPE interface. These two tasks execute in a mutually exclusive mode; SETUP has control during normal processing to receive user inputs directed to the DBMS, while the M/R task takes over during error processing (M/R mode) to handle error messages and responses directed to COPE. SETUP operates in a passive manner so that it is not obvious to the user that his inputs are going to COPE rather than directly to the DBMS. However, the presence of COPE becomes quite visible when the M/R task communicates with the user in English-like statements to give him specific information regarding errors, and to receive responses directed to COPE by the user attempting to rectify the detected errors.

BACKTRACK and LOCAL_DATA allow communication between different user contexts-- the LOCAL_DATA task associated with one user can activate the BACKTRACK task associated with another user. All communication with the DBMS is through LOCAL_DATA. BACKTRACK adjusts the context when the data base changes.

VERIFY receives input from SETUP, notifies M/R if there are errors, passes a completed and verified context to LOCAL_DATA if there are no errors, and is reactivated by BACKTRACK after the user corrects an error.

Information is passed between tasks via the user context. SETUP stores in the context the original and ICMDL versions of the user input and the list of applicable constraint rules. VERIFY adds the proof trees constructed during verification and pointers to data obtained from the DBMS stored by LOCAL_DATA.

As described earlier, a user context is intended to contain all of the user inputs which are interrelated in the sense that one cannot safely be passed on to the DBMS until all are known to satisfy the constraint rules. The group of user inputs contained in a given context is determined dynamically by the SETUP task which compares each new input to the existing contexts for this user, employing the concept of a relation cluster. Informally, a relation cluster is a group of relations which are interdependent because any changes in the tuples associated with one relation may affect the correctness of the tuples associated with another relation. Formal definitions for a relation cluster and an algorithm for determining one are given in Appendix C.

In the next section will be discussed a major component of COPE which is used by several of the tasks--the translator. Following that will be detailed descriptions of each task. To illustrate the methods employed, an example data base update is traced through the system. Stated in English, the update is: "Assign Smith as the captain of the Constellation."

5.2 The Translator

To accomplish the DBMS-language/ICMDL mapping, the translator uses the data base structure description (DBSD) and the relation templates (see chapter 4). The error checking performed during the translation from the DBMS language to equivalent ICMDL is syntactic, that is, the tests involve only the use of the data base structure description. With the help of a scanner and generator, errors such as non-existent files or fields, spelling errors in file names or field names, and illegal references to subfields can be detected. The translator will be implemented by standard techniques, such as the table driven parser of Aho and Ullman [1].

Example

The example update is expressed in Datalanguage as:

```
UPDATE ship WITH name EQ 'Constellation' captain = 'Smith';
```

and is translated into the ICMDL statement:

```
(ship s_name: 'Constellation' (:= s_captain 'Smith') t_stats: FREE)
```

The intended meaning of this statement is to cause the 'captain' argument to be set to the constant 'Smith' for each tuple in the 'ship' file with a 'name' argument containing the constant 'Constellation'. The argument for 'readings', which is represented in ICMDL by the argument 't_stats', does not need to be changed or examined, and is thus to be ignored. Because ICMDL requires all the arguments of a relation to be specified, 't_stats' is identified as one to be ignored by the keyword "FREE".

5.3 The SETUP TASK

Normal communication with the user is through SETUP. It invokes the translator to convert inputs from the language of the DBMS into ICMDL, and to perform preliminary error checking. If no errors are detected, SETUP then assigns the input to a user context, using the Context Determination algorithm given in Appendix C.

COPE defines the current context for a user as the context into which the last user input was assigned. If the new input does not belong to the current context COPE will guess that the user is changing contexts and will try to "close out" the current context; that is, it will check whether the context is completely verified and ready to pass on to the DBMS. If more information, such as additional updates, corrections, or clarifications, are needed from the user, those conditions are attached to the context. The context is suspended and passed to the M/R task. A suspended context may be reopened by the user at any time.

SETUP completes its processing of the user input by adding the associated set of constraint rules to the context. The templates associated with the input during translation contain pointers to the sets of constraint rules which are to be applied whenever an argument is to be updated. The individual rule sets are collected for each of the arguments being updated and are merged with the rule set already existing in the context.

There are two possible outcomes of the SETUP task:

- (1) If an error is detected by the translator, the error conditions are attached to the user input and the input is passed to the MESSAGE/RESPONSE task.
- (2) If the user input passes the checks in SETUP, its ICMDL and DBMS language representations and the list of applicable constraint rules are stored in the appropriate user context. The context is then passed to the VERIFY task.

Example

Assuming that the user had made no prior inputs, a single new context will be created for him. Only the 'captain' field is to be altered. Therefore, according to the template below, the set of rules which must be tested includes A1 and I1. These rules are stored in the user context, which is passed to the VERIFY task.

```
(RELATION ship TEMPLATE
  ((s_name:: ship_struct.name: (A1))
   (s_captain:: ship_struct.captain: (A1 I1))
   (t_stats:: (LIST OF (RECORD
     ((s_time:: stats.time: (A1 I1))
      (s_heading:: stats.heading: (A1)))))))
```

5.4 The VERIFY TASK

The purpose of the VERIFY task is to employ the constraint rules to perform the semantic error checks. Under the heuristic control of the search strategist, VERIFY applies the constraint rules selected by SETUP until either all the constraint rules have been applied or the search strategist causes termination of the checks or the user causes cancellation of the context.

Instantiation

Prior to checking a rule, VERIFY instantiates it, matching the variables of the formula against the values in the user inputs. If all variables can be given values, the formula is said to be fully instantiated, otherwise it is partially instantiated.

Example

SETUP added two rules to the context for VERIFY to check. Rule I1 requires that each ship have exactly one captain. In the example this will cause the 's_name' portion of the user input to be matched (or substituted for) the 's_name' portion of the pattern, with a similar match occurring for the 's_captain' fields. Wherever the pattern has an imbedded assignment, such as (:=s_name x), the variable involved is assigned the matching value, and all instances of that variable over the scope of its definition, which includes the formula portion, take on the same value. The resulting pattern and formula for the example rule are:

```

pattern: (ship $s_name: (:= 'Constellation' x)
          $s_captain: (:= 'Smith' y) $t_stats: FREE)

formula: ((TYPE z (SET OF CHARACTER))
          (AND (FOREACH x) (EQUAL (COUNT
                                   (SET UNION 'Smith' z)) 1)
               (ship $s_name: (s_name = 'Constellation')
                 $s_captain: (:= s_captain z) $t_stats: FREE)))

```

The variable z is to be the set of captains of the 'Constellation' which are specified in the DBMS data base, if any such exist. This must be instantiated by means of a retrieval from the DBMS.

Proof Tree Construction

VERIFY checks "the next" constraint rule (as ordered by the search strategist) to determine if it has been violated.

In the most general case, the formula portion, which is the semantic specification of the constraint, consists of multiple subformulae, each of which must be handled one at a time in the order in which they are specified. Thus the formula is essentially a program specified in a very general manner in an extended form of predicate calculus. As in predicate calculus, each subformula is a boolean statement which is composed of tuples called atoms connected with boolean operators.

For the purposes of the verification algorithm, the atoms of each formula must be divided into two types during the processing:

- Type 1 - Fully or partially instantiated atoms with a predicate that is executable. These are locally computable. Examples are: (EQ (COUNT z) 1), (GTEQ 'admiral' 'ensign').
- Type 2 - Atoms that have COPE relations as predicates. An example is: (ship \$s_name: (s_name = 'kennedy') \$s_captain: (:= s_captain y) \$t_stats: FREE).

The primary distinction between the two types of atoms is that Type 1 does not require any data to be retrieved from the DBMS, while Type 2 specifies a query which must ultimately be passed to the DBMS. Because it is assumed that the cost to both the DBMS and to COPE of making DBMS requests is high, COPE attempts to minimize the number and size of the DBMS requests by checking any Type 1 atoms it can before each request to the DBMS. A more formal description of the VERIFY task is provided by the Verification Algorithm which is given in Appendix C.

VERIFY keeps track of its progress by means of a proof tree for each subformula. The proof tree contains all of the information about the means by which each atom was satisfied. It is also used to record a failure which must be passed on to the M/R task. Each entry of a proof tree represents an atom which has been verified and contains pointers to the proof steps on which it depends. A list of supporting data facts is associated with each atom. The importance of the proof tree is explained further in the discussion of the BACKTRACK task.

A heuristic decision process is employed to determine which item to retrieve next from the data base. The heuristics will be based on such factors as:

- a. the particular relation involved;
- b. which variables need to be instantiated in order to be able to fully instantiate a locally computable atom;
- c. any other factors which would help minimize the amount of work that the DBMS must perform on this portion of the constraint rule.

Further consideration and determination of these heuristics will be undertaken during implementation of the VERIFY module.

Example

In the example, the formula is composed of one subformula, which in turn is composed of three atoms which are conjoined. These are:

- 1) (TYPE z (SET OF CHARACTER))
- 2) (FOREACH x) (EQUAL (COUNT (SET_UNION 'Smith' z)) 1)
- 3) (ship \$s_name: (s_name = 'Constellation')
 \$s_captain: (:= s_captain z) \$t_stats: FREE)

Therefore, in order for the rule to hold true, thus implying that no violation of the rule was found, all of the atoms of the subformula must be true.

Outcomes

There are three possible outcomes of the verification task. These outcomes determine which task will control the user context next:

- a. No rule failures - The search strategist causes termination because either all of the rules have been applied successfully, or there is no rule which can be applied within the current resource limitations on COPE. The context is considered verified and is forwarded to the LOCAL DATA task (section 5.5) which will pass the user inputs on to the DBMS.
- b. One or more rule failures - As a rule failure is detected by VERIFY, the associated proof tree is tagged with the error condition and the context is passed to the MESSAGE/RESPONSE task so that the user may be notified of the problem. VERIFY is suspended until the user corrects the situation.

- c. Undeterminable - If all the relevant rules have been applied to a context without any failures being detected, but there is not enough information in the context to complete verification, VERIFY is suspended. As in the case of a rule failure, the partially completed proof tree is tagged with the error condition, and the context is passed to the MESSAGE/RESPONSE task in order to communicate the problem to the user if necessary.

VERIFY may be resumed after suspension if further user inputs are received. If the suspension was due to insufficient information, subsequent inputs for that context will cause reactivation of VERIFY. If the suspension was due to a rule failure, the M/R task will aid the user in making changes to his previous inputs to correct the errors. However, changes to the user context imply the possibility that error checks already performed may no longer be valid. It may be necessary to unravel the proof trees past the point of the error and repeat some of the rules. This procedure is called backtracking and is described in section 5.6.

As is described in section 6, the search strategist controls the termination of VERIFY by deciding when to halt the application of rules to a context. If the search strategist terminates VERIFY with a completed context, the updates are ready to be entered in the data base.

Example

For the example user input, assume that in the application of rule 11 the VERIFY task discovered that the set of captains of the "Constellation" was of size 2 due to the occurrence of the tuple:

```
( ship s_name: "Constellation" s_captain: "Jones"
  t_stats: ( s_time: ( hour: 23 minute: 35 ) s_heading: 323))
```

in the data base. The result is that the rule fails, and the partial proof tree must be tagged with the error condition and the context passed to the M/R task.

5.5 The LOCAL_DATA TASK

The LOCAL_DATA task transmits user inputs to the DBMS. It also sends COPE queries generated by VERIFY to the DBMS, receives the responses, has them translated into ICMDL, and passes them back to VERIFY. In order to improve efficiency and monitor all access to data from user contexts, LOCAL_DATA maintains a local copy of each item from the main data base which is currently being referenced by any user context.

COPE assumes that only a small section of the DBMS data base is likely to be in use at any one time, so that space and time requirements for maintaining the local data base will not be a significant factor. Thus, by keeping within COPE the data currently in use, efficiency is increased since traffic between COPE and the DBMS is reduced and data access is faster. This efficiency consideration is important since COPE is meant to be invisible to the user and must not create bottlenecks due to its own processing, or add unduly to the burden on the DBMS. Since there is only one LOCAL_DATA task within COPE, regardless of the number of users, it will be shared by all users.

As a user context accesses a data item, this event is recorded in an "access list" which is associated with each datum. Thus when a user context is allowed to pass user inputs on to the DBMS, other user contexts which are referencing the same data can be located by means of the access list. An access list contains one entry for each user who is currently accessing that datum. Each entry points to a list of the instances in a proof tree where the user accessed the datum.

LOCAL_DATA is invoked by a VERIFY task when one of the following situations occurs:

- a) in the course of applying rules, a data base retrieval is necessary
- b) the search strategist determines that verification of a set of user inputs is complete.

Requests for data base retrievals are passed to LOCAL_DATA by VERIFY. If the requested item is found in the local data base, its access list is updated to include an entry for the accessing user context. The item is returned to VERIFY which is allowed to resume processing of the context. If the data cannot be found locally, a request in the language of the data base is formed and transmitted to the DBMS. If the request succeeds, the data is translated into ICMDL, added to the local data base along with an appropriately initialized access list, and returned to VERIFY. If the data base retrieval fails, VERIFY is notified of the failure by LOCAL_DATA.

LOCAL_DATA handles user inputs being passed on to the DBMS in a somewhat different manner. When a context is sufficiently verified, it is sent to LOCAL_DATA. Using the version of the user inputs expressed in the language of the DBMS, LOCAL_DATA transmits the user inputs to the DBMS. The access list for each item of the completed context is examined for two purposes. Since the local data base includes only those items currently being used, any data involved in the completing context but not being accessed by any other user contexts can now be deleted. However, if there are other user contexts referencing altered data items, those contexts must be checked in order to resolve any conflicts which may have occurred in the contexts as a result of the change to the data base. The checks for conflicts are handled by the BACKTRACK task.

Example

For the example input, it is necessary to query the DBMS to instantiate the variable y in the atom (ship \$s_captain: (:= s_captain y)), that is, to find the set of captains of the 'Constellation'. The DBMS returns the set containing 'Jones' as the value of y.

5.6 The BACKTRACK TASK

One of the most complex situations which COPE must handle is that which occurs when an item in a user context is altered while being used in error checking. A user context contains a list of the rules applied and being applied in the form of a collection of proof trees, user supplied data, and

data obtained from the DBMS. However, the latter two entities are subject to change, and modifications in the local or DBMS data bases imply the possibility that tests already performed in a partially completed context may no longer be valid. To avoid a violation of the constraint rules, it is necessary to find any conflicts created in the open contexts as a result of the alterations of the data and redo some or all of the error checks in those user contexts. The BACKTRACK task accomplishes the unraveling of proof trees.

There are two situations that can result in a change of an item in a user context. First, a single user may create a conflict by changing an item in order to correct an error. Second, in a multi-user environment, one user may update items being referenced or changed by another user.

An example illustrating items being changed in a single user situation is as follows:

- a. The user enters a collection of inputs which cause the VERIFY task to apply constraint rules.
- b. VERIFY detects a rule violation and passes the message to the MESSAGE/RESPONSE task.
- c. The M/R task informs the user of the violation and advises a correction.
- d. The user specifies that a previous input be deleted and a new input substituted.

In order to validate the substitution of user inputs, references in the context to the old input must be removed and the new input added and verified.

An example of the multi-user situation is given by the following scenario involving two users:

- a. User A and User B each enter inputs involving the same portion of the data base.
- b. User A's context completes and the LOCAL_DATA task updates the data base accordingly.
- c. User B's context is now in conflict with the data base so that rule violations occur.

All proof trees in User B's context must be unraveled past the point of first use of any modified data items and the constraint rules must be reapplied.

The BACKTRACK task is invoked by either the LOCAL_DATA task or the M/R task, depending on the cause of the alteration of the context. It is started by LOCAL_DATA when another user has changed the data base and by the M/R task when an error correction involving the deletion, modification, or replacement of an input in a user context has occurred.

The backtracking algorithm given in Appendix C specifies how BACKTRACK examines and modifies a context to return it to a conflict-free state. At its termination, unless a rule failure was discovered, VERIFY is unblocked and allowed to resume checking of the context. If a rule failed, the user is notified by the MESSAGE/RESPONSE task.

Example

The example user input represents a single user error condition. The user's input was in error because "Jones" was already assigned as the captain of the "Constellation". The user knows that "Jones" is to retire, and "Smith" has been assigned as his replacement. Therefore "Jones" should have been removed from the position as captain of the "Constellation" thus leaving that position open for reassignment. Therefore, when the user is informed by the MESSAGE/RESPONSE task why his input is considered to be in error, he enters an additional input which is translated into the ICMDL statement:

```
( ship s_name: 'Constellation' (:= s_captain: UNASSIGNED )
  t_stats: FREE )
```

This is passed to the BACKTRACK task which understands that the new user input must be treated as if it occurred prior to the originally presented input. BACKTRACK will unravel the partial proof tree in order to remove the fact specifying "Jones" as the captain of the "Constellation" and replace that fact with one specifying that the captain is unassigned. The constraint rule will be satisfied, thus permitting the completed context to be passed to LOCAL_DATA, which will send the original user input on to the DBMS.

5.7 The MESSAGE/RESPONSE (M/R) TASK

The MESSAGE/RESPONSE (M/R) task handles error messages and user responses directed to COPE. The interactive mode (M/R mode) of COPE is designed to provide the user with as much assistance as possible to allow correction of errors. If an error is such that COPE has an idea of the required correction, the M/R task will prompt the user accordingly. The M/R task also allows the user to ask questions regarding the operation of COPE and how an error was detected. Since the proof trees and rule list were saved in the context, the M/R task is able to provide a derivation of the error condition. The M/R task is also able to answer specific questions about anything in the user context or the constraint rules and templates of the Conceptual Model.

SETUP passes M/R two types of entries: inputs with syntactic errors detected during translation, and suspended user contexts which represent incomplete transactions. VERIFY passes contexts having errors discovered when applying the constraint rules. COPE does not output messages as soon as potential errors are detected because subsequent user inputs may correct the error and thus eliminate the need to notify the user. Errors are saved on a queue until one of the following events occurs:

- 1) The user submits a special input requesting to enter M/R mode to examine the error messages;

- 2) The search strategist forces the user into M/R mode because the system performance is degrading due to having too many open contexts; or,
- 3) User inputs correct the error situation in a suspended context so that processing by VERIFY may continue.

When the system enters M/R mode, messages are formed using the information provided to the M/R task and output one by one to the user. The contexts are released to the appropriate task as they are fixed. In the third case mentioned above, the error situation is corrected without aid from the M/R task, allowing the context to be released from M/R and returned to VERIFY for continuation of the checking. These actions are explained further below.

The user must respond to every COPE message. Since SETUP is blocked during M/R mode, all user inputs are received by the M/R task. Because COPE was not intended to be a natural language query system, it recognizes only a small set of non-ICMDL user responses. However, prompts by the M/R task direct the user in making valid replies.

In addition to communicating with the user, the M/R task has the responsibility of performing the requested corrections and passing the corrected context or input to the appropriate task. These operations differ, depending on the task which detected the error and the type of correction made. Errors detected by SETUP are strictly syntactic; therefore, they do not involve the constraint rules and can be handled independently of any context. For this reason, a context does not have to be suspended due to errors detected by SETUP, and multiple inputs with syntactic errors may be handled simultaneously. Only the input in error, not the context, is sent to the M/R task. The user has two choices of methods to correct errors detected by SETUP:

- a) Field substitution - The requested change is substituted in the input, which is then returned to SETUP at the place from which the M/R task was invoked, so that translation may continue.
- b) Replacement - The old input and any partial translation are deleted and translation of the corrected input is started from the beginning by COPE.

The errors detected by VERIFY are violations of constraint rules and are associated with a context, not just a single input. Therefore, verification attempts for that user context are suspended until the situation is corrected. The suspended context with error conditions attached is passed to the M/R task. Corrections are handled in the following manner:

- a) Abort - Delete the entire context and all associated user inputs.
- b) Deletion - BACKTRACK is invoked to delete the input and to undo everything associated with that input.
- c) Modification - The substitution is made to the specific field of the input and the context is passed to BACKTRACK which decides what unraveling must be done.

- d) Addition - SETUP is invoked with the new input and context in order to perform translation and determine the relevant constraint rules. VERIFY is then invoked with the translated input and context.
- e) Replacement - SETUP is called to translate the new input and to determine the rules. BACKTRACK is then invoked with the context and the translated input intended to replace the old item.

The M/R task relinquishes control of the user/COPE interface when either all the errors have been handled, or the user requests to leave M/R mode.

Example

For the example, the M/R task may output a message such as:

The ship with name 'Constellation' has 'Jones' assigned as captain.
Please reassign 'Jones'.

The user could respond with the statement:

UPDATE person WITH name EQ 'Jones' duty = NULL; status = 'inactive';

which would be sent by M/R to SETUP for translation and further processing.

6. THE SEARCH STRATEGIST

Introduction

Global control of the COPE system is provided by the search strategist. Its role is to ensure the greatest level of error detection possible for each user without degradation of the performance of the DBMS/COPE system.

Overloading of the DBMS and bottlenecks in the DBMS/COPE communication are problems which must be prevented by the search strategist. This is particularly important in a multi-user environment in which many users pass DBMS requests through COPE at the same time. The demands placed on COPE could often exceed the ability of the tasks to perform all possible error checks on all user inputs without seriously slowing down the overall response time for the DBMS/COPE. By means of heuristic guidance, the search strategist must allocate the limited resources of COPE to do the best possible job for each user.

The search strategist must constantly adapt to the conditions of the environment, taking into account:

- * DBMS request load
- * COPE work load
- * costs of error detection
- * costs of failing to detect errors

To do so, it can adjust:

- * which constraint rules are applied
- * the resource allocation
- * the cost threshold

Constraint rules have been discussed in section 4. The resources which can be allocated to users include space, time, communication channel and DBMS resources. The cost threshold is the value which the cost parameter of a selected rule may not exceed. It is dynamically adjusted to reflect the current load on the DBMS/COPE system. Only those applicable rules which have a cost less than or equal to the cost threshold will actually be applied. Since the cost threshold may change during the processing of a user context, the rule is only temporarily blocked and can be reconsidered anytime before the context is closed.

There are no hard-and-fast rules which give the search strategist the ability to make clear cut decisions. All of these factors are not only subjective in their measurement at any given time, but are also subject to rapid changes. Therefore the search strategist must employ a set of heuristic guides keyed to the performance and load information available in order to make the necessary decisions.

Cost versus value

The purpose of COPE is to provide both the users and the DBMS with a service, namely preventing the introduction of errors into a data base. The value of this service is measured in terms of the importance to the users of having correct data and the cost to the user of correcting an erroneous data base later. Because the costs of erroneous data are often high, the corresponding value for preventing errors is also high. On the other hand, the detection of errors does not come free; there is a cost for the processing which COPE performs. That cost is measured both in terms of the additional hardware and software costs for creating and maintaining COPE, and the interference with the flow of requests from the user to the DBMS. It is essential that this cost be kept in balance with the value being obtained from error detection.

For some constraint rules, the entire cost is incurred by processing data within the COPE system; for others, there is an additional cost associated with obtaining information from the DBMS. In the former case, the cost is strictly a function of the efficiency and effectiveness of COPE, while in the latter case, the cost must include the DBMS time and communication channel resources required to pass requests to the DBMS and to receive the responses. In addition, these checks require extra COPE resources for ICMDL-DBMS language translation. The extra cost is often offset by the greater value of the more extensive checks.

Monitoring system performance

The search strategist monitors and regulates the overall system performance by dynamically adjusting a) the cost threshold value and b) the resource allocations. This section discusses guidelines for determining the system load and making the adjustments.

A method such as that which follows will be used to detect changes in the system load which require adjustments of the cost threshold and/or resource allocation. The search strategist divides time into major time units (MTU), which will typically have a duration of five to ten seconds. At the end of each major time unit it examines the number of input user requests waiting to be processed by COPE, the rate of change in the number of input requests over the previous five MTU's, the number of approved requests passed on to the DBMS in the previous MTU, and the rate of change of the number of approved requests over the previous five MTU's. If the number of approved requests approximates the number of input requests, then no adjustments of either the cost threshold or the resource allocations are required. If the rate of change of the number of input requests is much greater or much less than the rate of change of the number of approved requests, then either the cost threshold or the resource allocation or both must be changed.

Changes in the threshold will have a more immediate effect than changes in the resource allocation. A reduction in the threshold value will immediately restrict the selection of constraint rules, while a reduction in the resource

allocation will only effect a context when it has a scarcity of resources. A raising of the threshold value will immediately permit previously blocked rules to be reconsidered, while an increase in the resource allocation will only immediately affect those contexts which were about to run out of resources. Thus, sudden but brief increases in the volume of incoming user requests can be accommodated by temporary changes in the cost threshold.

When there is a significant change in the relative rates of increase or decrease in the input or output volume, then the resource allocation must be altered to reflect this change in demand on the system. This will allow COPE to reduce the overall effort which can be expended on any given user context when the volume of user requests is high and to increase the efforts for each context when the volume is low. Thus more checking will be performed when possible, but not at the expense of creating a bottleneck in the overall throughput of the system.

Another factor which must be controlled is the load which COPE places on the DBMS. In situations where COPE cannot complete the checking of a constraint rule without requests to the DBMS, the exact time to make the DBMS request is rarely fixed. Requests can be carried along until no further processing can be done without actual DBMS supplied data. Thus COPE can smooth out the load on the DBMS by postponing some requests until they are essential.

Heuristic rule control

The heuristic rule control mechanism determines which applicable constraint rule should be applied next in each open user context by performing computations on rule components. These are:

- a) the cost - the effort involved in applying the rule. This includes the cost of accessing the data base and the cost of resources.
- b) the payoff - the likelihood that an error will be detected by applying this rule or the likelihood that failure to apply this rule will result in introduction of an error into the data base.

For each user, the search strategist is concerned with obtaining the most checking, that is, minimizing the likelihood that errors will enter the data base. Thus, the first heuristic (A) is to select the rule which has the highest payoff. However, the search strategist requires that any rule applied must have a cost which is less then or equal to the cost threshold. Therefore, the next attempt (heuristic B) is to select the rule which has the highest payoff, with a cost within the bounds of the cost threshold.

The lack of sufficient resources may prohibit execution of the rule chosen by heuristic B. One of the following heuristics may be adopted in its place:

- a) Continue applying rules chosen by heuristic B until resources are exhausted, then terminate checking of this user context.

- b) If adequate resources are available for the rule chosen by heuristic B, then apply it. Otherwise, discard that rule and try heuristic B again.

In the evaluation of COPE, these and other heuristic algorithms will be tested and compared.

Areas for future research

There are a number of subtle issues concerning the determination of the value of applying a particular constraint rule in a given situation. These issues are beyond this phase of the COPE project, but they are included here as suggestions for future research.

- a) Influence of current situation

In some cases it may not be desirable that all the rules with a particular threshold value are blocked at the same time. For instance, if a crisis situation has just developed in the South Pacific, it may be desired that the system devote most of its error checking resources to maintaining error-free data about the various elements of the data base associated with that crisis. This implies that the rules associated with the "ship" relation would be applied differently if the particular ship were located in the South Pacific rather than the Atlantic.

- b) Effect of different users

Different sources of data will often have different levels of credibility for their information and different probabilities of errors of a particular type. Therefore it may be appropriate to adjust the payoff and cost measures of the rules depending on the source of the data. Similarly, the search strategist may be designed to handle users with different priority levels. Thus some users might be delayed longer than others and some might be given larger resource allotments than others.

- c) Interaction of errors

Humans can often detect anomalies by noticing conflicts between one piece of data and another. This is exactly the type of knowledge which the constraint rules are intended to provide to COPE. But when multiple errors occur, the conflicts may disappear within the data base, although a larger view of the world would enable them to be detected. Therefore, when COPE is gradually turned off during a crisis situation, there is no way to determine whether the static application of COPE after the crisis will fail to detect major errors in the data base. An understanding of the interaction of compound errors is essential to the proper control of the graceful degradation of COPE and recovery from these errors after the crisis has subsided.

7.Integrity Checking Scenarios

Presented in this section are scenarios which illustrate user-COPE interaction for several typical situations. The examples are centered around a Navy-oriented data base. Datalanguage is used as the DBMS language.

First, the data base is described as it would be declared in Datalanguage. Next, elements of the Conceptual Framework corresponding to this data base are presented in their ICMDL representation. With this basis, scenarios dealing with such issues as simple error detection and simultaneous updates are presented.

7.1 The Data Base

This section consists of the Datalanguage statements used to define the logical structures of the data base which are used in the scenarios.

```
create bluefile;
create bluefile.ship file list, p=eof, chapter
  ship_struct structure
    name string (,20)
    captain string (,20)
    shipclas string (,20)
    port string (,20)
    destination string (,20)
    assignment string (,20)
    readings_count integer
    readings_list (,10,100), c=readings_count
    stats structure
      time structure
        year integer
        month integer
        day integer
        hour integer
        minutes integer
      end
    position structure
      lat_degrees integer
      lat_minutes integer
      lat_direction string (,1)
      long_degrees integer
      long_minutes integer
      long_direction string (,1)
    end
    fuel_amt integer
    speed integer
    heading integer
    eta structure
```

```

        year integer
        month integer
        day integer
        hour integer
        minutes integer
    end
    dest_pos string (,12)
end
end;

create bluefile.installation file list, p=eof, chapter
installation_struct structure
    name string (,20)
    position structure
        lat_degrees integer
        lat_minutes integer
        lat_direction string (,1)
        long_degrees integer
        long_minutes integer
        long_direction string (,1)
    end
end;

create bluefile.shipclass file list, p=eof, chapter
shipclass_struct structure
    class string (,20)
    fuel_cap integer
    length integer
    maxspeed integer
    galls_per_hr integer
    ships_count integer
    ships_list (,20,60), c=ships_count
    names structure
        name string (,20)
    end
end;

create bluefile.person file list, p=eof, chapter
person_struct structure
    name string (,20)
    rank string (,12)
    platform string (,20)
    jobclass string (,20)
    status string (,8)
    date_assigned structure
        year integer
        month integer
        day integer
    end
end;;

```

7.2 The Conceptual Framework

As discussed in section 4, the Conceptual Framework consists of four parts:

- 1) the data base structure description
- 2) the relation templates
- 3) the semantic dictionary
- 4) the constraint rules

This section contains the knowledge (expressed in ICMDL) which would be supplied to COPE by a human data base expert.

The Data Base Structure Description

Presented below is the ICMDL version of the database which was given in Datalanguage in section 7.1

```
((FILE ship (LIST WITH TERMINATOR = EOF OF
(TYPE ship_struct (RECORD
  ((name: (TYPE name (ARRAY (1 20) OF CHAR)))
  (captain: name)
  (shipclas: name)
  (port: name)
  (destination: name)
  (assignment: name)
  (readings_count: integer)
  (readings: (LIST WITH SIZE = readings_count OF
    (TYPE stats (RECORD
      ((time: (TYPE time (RECORD
        ((year: integer)
        (month: integer)
        (day: integer)
        (hour: integer)
        (minutes: integer))))))
      (position: (TYPE position (RECORD
        ((lat_degrees: integer)
        (lat_minutes: integer)
        (lat_direction: CHAR)
        (long_degrees: integer)
        (long_minutes: integer)
        (long_direction: CHAR))))))
      (fuel_amt: integer)
      (speed: integer)
      (heading: integer)
      (eta: time)
      (dest_pos: position))))))
  ))))
```

```
(FILE installation (LIST WITH TERMINATOR = EOF OF
(TYPE installation_struct (RECORD
  ((name: name)
  (position: position))))))
```

```
(FILE shipclass (LIST WITH TERMINATOR = EOF OF
(TYPE shipclass_struct (RECORD
  ((class: name)
  (fuel_cap: integer)
  (length: integer)
  (maxspeed: integer)
  (galls_per_hr: integer)
  (ships_count: integer)
  (ships: (LIST WITH SIZE = ships_count OF
    (TYPE names (RECORD
      ((name: name))))))))))
```

```
(FILE person (LIST WITH TERMINATOR = EOF OF
(TYPE person_struct (RECORD
  ((name: name)
  (rank: (TYPE rank (ARRAY (1 12) OF CHAR)))
  (platform: name)
  (jobclass: name)
  (status: (TYPE status (ARRAY (1 8) OF CHAR)))
  (date_assigned: (RECORD
    ((year: integer)
    (month: integer)
    (day: integer))))))))
```

The Relation Templates

The templates given in this section define the COPE relations and the mapping of each argument of each relation to the corresponding field in the data base. They also contain lists of the applicable constraint rules.

```
((RELATION ship TEMPLATE
  ((s_name:: ship_struct.name: (A1))
  (s_captain:: ship_struct.captain: (A1 I1))
  (s_shipclas:: ship_struct.shipclas: (A1))
  (s_port:: ship_struct.port: (A1))
  (s_dest:: ship_struct.destination: (A1 I10))
  (s_assignment:: ship_struct.assignment: (A1 I6))
  (t_stats:: (LIST OF (RECORD
    ((s_time:: stats.time: (A1 I11))
    (s_position:: stats.position: (A1 I14))
    (s_fuel_amt:: stats.fuel_amt: (A1 I8))
    (s_speed:: stats.speed: (A1 I7))
    (s_heading:: stats.heading: (A1))
    (s_eta:: stats.eta: (A1 I9 I10 I12))
    (s_dest_pos:: stats.dest_pos: (A1)))))))
```

(RELATION installation TEMPLATE

((i_name:: installation_struct.name: (A2))
(i_position:: installation_struct.position: (A2))))

(RELATION shipclass TEMPLATE

((sc_class:: shipclass_struct.class: (A3))
(sc_fuel_cap:: shipclass_struct.fuel_cap: (A3))
(sc_length:: shipclass_struct.length: (A3))
(sc_maxspeed:: shipclass_struct.maxspeed: (A3))
(sc_galls_per_hr:: shipclass_struct.galls_per_hr: (A3))
(sc_ships:: (LIST OF (RECORD
((sc_name:: names.name: (A3))))))))

(RELATION person TEMPLATE

((p_name:: person_struct.name: (A4))
(p_rank:: person_struct.rank: (A4))
(p_platform:: person_struct.platform: (A4 I2 I3))
(p_jobclass:: person_struct.jobclass: (A4 I2 I4))
(p_status:: person_struct.status: (A4))
(p_date_assigned:: person_struct.date_assigned: (A4 I2 I5 I13))))

The Semantic Dictionary

This section contains the hierarchy of collections (semantic classes) of items in the example data base. Note, for instance, that the class of ship captains, 'scaptain', is a subclass of the class of person names, 'pname'.

((ISA shipname (SET := (PROJECT (ship (: s_name))))))
(ISA scaptain (SET := (PROJECT (person (: p_name) (p_rank = 'captain')))))
(ISA sclass (SET := (PROJECT (shipclass (: sc_class))))
(ISA sbase (SET := (PROJECT (installation (: i_name))))
(ISA sassignment (SET := (PROJECT (ship (: s_assignment))))
(ISA stime (SET := (PROJECT (ship (: s_time))))
(ISA sposition (SET := (PROJECT (ship (: s_position))))
(ISA sfuel_amt (SET := (PROJECT (ship (: s_fuel_amt))))
(ISA sspeed (SET := (PROJECT (ship (: s_speed))))
(ISA sheading (SET := (PROJECT (ship (: s_heading))))
(ISA seta (SET := (PROJECT (ship (: s_eta))))
(ISA sdest_pos (SET := (PROJECT (ship (: s_dest_pos))))
(ISA iposition (SET := (PROJECT (installation (: i_position))))
(ISA scfuel_cap (SET := (PROJECT (shipclass (: sc_fuel_cap))))
(ISA sclength (SET := (PROJECT (shipclass (: sc_length))))
(ISA scmaxspeed (SET := (PROJECT (shipclass (: sc_maxspeed))))
(ISA scgalls_per_hr (SET := (PROJECT (shipclass (: sc_galls_per_hr))))
(ISA scname (SET := (PROJECT (shipclass (: sc_name))))
(ISA pname (SET := (PROJECT (person (: p_name))))
(ISA prank (LIST := (seaman ensign captain general admiral)))
(ISA pplatform (SET UNION (SET := (PROJECT (ship (: s_name)))
(SET := (PROJECT (installation (: i_name))))))
(ISA pjobclass (SET := (PROJECT (person (: p_jobclass))))
(ISA pstatus (SET := (active inactive)))
(ISA pdate_assigned (SET := (PROJECT (person (: p_date_assigned))))))

The Constraint Rules

The integrity constraint rules referenced in the relation templates are listed below. The rule name, pattern, and formula are given for each rule; the action, cost, and payoff are omitted. The first set of rules (A1 through A4) corresponds to the ISA hierarchy and tests for semantic class membership.

- * The ship relation describes a ship by specifying its name, captain, class, port and destination bases, fuel amount, speed, heading, estimated time of arrival, and destination position.

```
(RULE A1 (ship $s_name: (:= s_name x1) $s_captain: (:= s_captain x2)
  $s_shipclas: (:= s_shipclas x3) $s_port: (:= s_port x4)
  $s_destination: (:= s_destination x5) $s_assignment:
    (:= s_assignment x6) $s_time: (:= s_time x7) $s_position:
    (:= s_position x8) $s_fuel_amt: (:= s_fuel_amt x9) $s_speed:
    (:= s_speed x10) $s_heading: (:= s_heading x11) $s_eta: (:= s_eta x12)
  $s_dest_pos: (:= s_dest_pos x13))
  (ship $s_name: (s_name = shipname) $s_captain: (s_captain = scaptain)
  $s_shipclas: (s_shipclas = sclass) $s_port: (s_port = sbase)
  $s_destination: (s_destination = sbase) $s_assignment:
    (s_assignment = sassignment) $s_time: (s_time = stime) $s_position:
    (s_position = sposition) $s_fuel_amt: (s_fuel_amt = sfuel_amt)
  $s_speed: (s_speed = sspeed) $s_heading: (s_heading = sheading)
  $s_eta: (s_eta = seta) $s_dest_pos: (s_dest_pos = sdest_pos)))
```

- * The installation relation consists of the names and geographical positions of naval bases.

```
(RULE A2 (installation $i_name: (:= i_name x1)
  $i_position: (:= i_position x2))
  (installation $i_name: (i_name = sbase) $i_position:
    (i_position = iposition)))
```

- * The shipclass relation contains the attributes of different types of ships. These attributes are class, fuel capacity, length, maximum speed, gallons per hour, and the names of ships of that class.

```
(RULE A3 (shipclass $sc_class: (:= sc_class x1) $sc_fuel_cap:
  (:= sc_fuel_cap x2) $sc_length: (:= sc_length x3) $sc_maxspeed:
  (:= sc_maxspeed x4) $sc_galls_per_hr: (:= sc_galls_per_hr x5)
  $sc_name: (:= sc_name x6))
  (shipclass $sc_class: (sc_class = sclass) $sc_fuel_cap:
    (sc_fuel_cap = scfuel_cap) $sc_length: (sc_length = sclength)
  $sc_maxspeed: (sc_maxspeed = scmaxspeed) $sc_galls_per_hr:
    (sc_galls_per_hr = scgalls_per_hr) $sc_name: (sc_name = scname)))
```

- * The person relation specifies the name, rank, platform, jobclass, status, and date of assignment.

```
(RULE A4 (person $p_name: (:= p_name x1) $p_rank: (:= p_rank x2)
  $p_platform: (:= p_platform x3) $p_jobclass: (:= p_jobclass x4)
  $p_status: (:= p_status x5) $p_date_assigned: (:= p_date_assigned x6))
  (person $p_name: (p_name = pname) $p_rank: (p_rank = prank)
  $p_platform: (p_platform = pplatform) $p_jobclass:
  (p_jobclass = pjobclass) $p_status: (p_status = pstatus)
  $p_date_assigned: (p_date_assigned = pdate_assigned)))
```

- * A ship has exactly one captain.

```
(RULE I1 (ship $s_name: (:= s_name x) $s_captain: (:= s_captain y))
  $s_shipclas: FREE $s_port: FREE $s_destination: FREE $s_assignment:
  FREE $s_time: FREE $s_position: FREE $s_fuel_amt: FREE $s_speed:
  FREE $s_heading: FREE $s_eta: FREE $s_dest_pos: FREE)
  (((TYPE z (SET OF CHARACTER)))
  (AND (FOREACH x) (EQUAL (COUNT (SET UNION y z)) 1)
    (ship $s_name: (s_name = x) $s_captain: (:= s_captain z)
    $s_shipclas: FREE $s_port: FREE $s_destination: FREE
    $s_assignment: FREE $s_time: FREE $s_position: FREE
    $s_fuel_amt: FREE $s_speed: FREE $s_heading: FREE $s_eta: FREE
    $s_dest_pos: FREE))))
```

- * A person must be in an active status in order to have his platform, jobclass, or date_assigned changed.

```
(RULE I2 (person $p_name: (:= p_name x) $p_rank: FREE $p_platform: FREE
  $p_jobclass: FREE $p_status: FREE $p_date_assigned: FREE)
  ((person $p_name: (p_name = x) $p_rank: FREE $p_platform: FREE
  $p_jobclass: FREE $status: (status = 'active') $p_date_assigned:
  FREE)))
```

- * A person can be assigned to at most one platform.

```
(RULE I3 (person $p_name: (:= p_name x) $p_rank: FREE $p_platform:
  (:= p_platform y) $p_jobclass: FREE $p_status: FREE
  $p_date_assigned: FREE)
  (((TYPE z (SET OF CHARACTER)))
  (AND (FOREACH x) (LTEQ (COUNT (SET UNION y z)) 1)
    (person $p_name: (p_name = x) $p_rank: FREE $p_platform:
    (:= p_platform z) $p_jobclass: FREE $p_status: FREE
    $p_date_assigned: FREE))))
```

- * A person whose job is chief navigator must have rank greater than or equal to ensign.

```
(RULE I4 (person $p_name: (:= p_name x) $p_rank: FREE $p_platform: FREE
  $p_jobclass: FREE $p_status: FREE $p_date_assigned: FREE)
  (((TYPE y (SET OF CHARACTER)))
  (AND (FOREACH x) (GTEQ y 'ensign')
    (person $p_name: (p_name = x) $p_rank: (:= p_rank y)
      $p_platform: FREE $p_jobclass: (p_jobclass = 'chief_navigator')
      $p_status: FREE $p_date_assigned: FREE))))
```

- * A person cannot be reassigned within 30 days.

```
(RULE I5 (person $p_name: (:= p_name x) $p_rank: FREE $p_platform: FREE
  $p_jobclass: FREE $p_status: FREE $p_date_assigned:
  (:= p_date_assigned y))
  (((TYPE old_date_assigned p_date_assigned))
  ((GTEQ y (PLUS old_date_assigned 30)
    (person $p_name: (p_name = x) $p_rank: FREE $p_platform: FREE
      $p_jobclass: FREE $p_status: FREE $p_date_assigned:
      (:= (p_date_assigned = [y]) old_date_assigned))))))
```

To aid readability, only the parameters which are not FREE will be given in the remainder of the constraint rules.

- * There must be an aircraft carrier assigned to each zone.

```
(RULE I6 (ship $s_name: (:= s_name x) $s_assignment: (:= s_assignment y))
  (((TYPE z (SET OF CHARACTER)))
  (OR
    (ship $s_name: (s_name = x) $s_assignment: (s_assignment = y))
  (AND
    (ship $s_name: (s_name = x) $s_assignment: (:= s_assignment z))
    (ship $s_name: (s_name = [x]) $s_shipclas:
      (s_shipclas = 'aircraft_carrier') $s_assignment:
      (s_assignment = z))))))
```

- * A ship's speed cannot exceed the maximum for that type.

```
(RULE I7 (ship $s_name: (:= s_name x) $s_speed: (:= s_speed y))
  (((TYPE z (SET OF CHARACTER)) (TYPE w integer))
  (AND (LTEQ y w)
    (ship $s_name: (s_name = x) $s_shipclas: (:= s_shipclas z))
    (shipclass $sc_class: (sc_class = z) $sc_maxspeed:
      (:= sc_maxspeed w))))
```

- * A ship's fuel amount cannot exceed the maximum for that type.

```
(RULE I8 (ship $s_name: (:= s_name x) $s_fuel_amt: (:= s_fuel_amt y))
  (((TYPE z (SET OF CHARACTER)) (TYPE w integer))
  (AND (LTEQ y w)
    (ship $s_name: (s_name = x) $s_shipclas: (:= s_shipclas z))
    (shipclass $sc_class: (sc_class = z) $sc_fuel_cap:
      (:= sc_fuel_cap w))))))
```

- * The expected time of arrival (eta) cannot be less than the calculated time of arrival which is the current time plus the distance to be traveled divided by speed.

```
(RULE I9 (ship $s_name: (:= s_name x) $s_eta: (:= s_eta y))
  (((TYPE t s_time) (TYPE z s_position) (TYPE w s_position)
  (TYPE v integer))
  (AND (GTEQ y (PLUS t (DIVIDE (DISTANCE z w) v)))
    (ship $s_name: (s_name = x) $s_time: (:= s_time t)
    $s_position: (:= s_position z) $s_dest_pos: (:= s_dest_pos w)
    $s_speed: (:= s_speed v))))))
```

- * A ship's fuel amount must be greater than or equal to the time to travel times the gallons per hour.

```
(RULE I10 (ship $s_name: (:= s_name x))
  (((TYPE v integer) (TYPE y time) (TYPE z time) (TYPE w integer)
  (TYPE t (SET OF CHARACTER)))
  (AND (GTEQ v (MULTIPLY (SUBTR y z) w))
    (ship $s_name: (s_name = x) $s_shipclas: (:= s_shipclas t)
    $s_fuel_amt: (:= s_fuel_amt v) $s_eta: (:= s_eta y)
    $s_time: (:= s_time z))
    (shipclass $sc_class: (sc_class = t) $sc_galls_per_hr:
      (:= sc_galls_per_hr w))))))
```

The following rules reference the executable predicates CHECKTIME, CHECKDATE, and CHECKPOS1 which invoke routines to test for legal time, date, and position values.

- * Check that ship.time is a legal time.

```
(RULE I11 (ship $s_time: (:= $s_time y)) (CHECKTIME y))
```

- * Check that ship.eta is a legal time.

```
(RULE I12 (ship $s_eta: (:= $s_eta x)) (CHECKTIME x))
```

- * Check that person.date_assigned is a legal data.

(RULE I13 (person \$p_date_assigned: (:= \$p_date_assigned x)) (CHECKDATE x))

- * Check that ship.position is a legal value.

(RULE I14 (ship \$s_position: (:= \$s_position x)) (CHECKPOS1 x))

7.3 Sample Scenarios

Each scenario described below will include a brief statement of the situation involved, the user-COPE dialogue, and a discussion of the step-by-step processing performed by COPE. According to the definition of ICMDL, each parameter must be explicitly mentioned in the constraint rule. However, in these examples only the relevant (non-FREE) parameters will be shown.

The data base expert specifies the "action" to be performed when a constraint rule violation is detected. Included in the action is the error message to be output by the M/R task. In the following scenarios, the COPE messages are part of the action for the rules listed in section 7.2. Acting as the data base experts, we have written these messages to be specific and English-like.

Scenario showing COPE detection of simple errors

In this example, COPE recognizes and corrects a spelling error in a user's update request. COPE also detects an error in the semantic class membership of another datum and asks the user for help in correcting the problem.

Current data base state:

The Kennedy is an aircraft carrier.

The Kennedy is stationed at Portland.

The set of installations is (Portland, San Diego, Honolulu,
San Francisco).

```
(ship s_name: 'Kennedy' s_shipclas: 'aircraft_carrier' s_port: 'Portland')
(shipclass sc_class: 'aircraft_carrier' sc_maxspeed: 35 sc_name: 'Kennedy')
(installation i_name: (SET := ('Portland' 'San_Diego' 'Honolulu'
'San_Francisco'))))
```

```
USER A - OPEN %TOP.navyfile.ship WRITE;
        UPDATE ship WITH name EQ 'Kenedy'
        destination='San_Jose';
```

```
COPE - No ship has name 'Kenedy'.
      Do you mean 'Kennedy'?
```

```
USER A - YES
```

```
COPE - No installation has name 'San_Jose'.
      Would you like to replace the name?
```

```
USER A - San_Diego
```

The translator converts the above update request into the ICMDL statement:

```
(ship s_name: 'Kenedy' (:= s_destination: 'San_Jose'))).
```

No syntactic errors are detected so COPE determines the user context and consults the template for 'ship' to obtain the set of applicable constraint rules. The rule A1 is applied to verify the semantic class membership of the argument values in the user input. For the argument 'name', the rule term:

```
$s_name: (s_name = shipname)
```

is used and it refers to the ISA statement:

```
(ISA shipname (SET := (PROJECT (ship (:= s_name)))))).
```

The set of shipnames is thus retrieved from the data base by the LOCAL DATA task. The set does not include 'Kenedy'; however, it does include 'Kennedy' which the spelling checker determines is sufficiently similar that a spelling error is likely. The correction is suggested to the user by the M/R task, the user agrees, and the substitution is made.

Rule A1 is then applied to the remaining argument, 'destination'. The term:

```
$s_destination: (s_destination = sbase)
```

refers to the ISA statement:

```
(ISA sbase (SET := (PROJECT (installation (:= i_name)))))).
```

The set of installation names retrieved from the data base does not contain 'San_Jose' and there is no name close enough to suspect a spelling error. The M/R task asks the user to replace the incorrect datum, and the user substitutes 'San_Diego' to create the input:

```
(ship s_name: 'Kennedy' (:= s_destination 'San_Diego'))).
```

The input has now been verified according to the example Conceptual Framework.

Scenario showing sharing of data between two users

This scenario involves two users who are simultaneously accessing and updating the same data. One user will alter data without the other's knowledge.

Current data base state:

The Enterprise is an aircraft carrier.

Commodore Perry has been captain of the Enterprise since 6-6-71.

Fisher has rank of seaman, has been stationed at San Diego Naval Base since 10-2-74, and is a clerk.

Cooper has rank of seaman, has been stationed at San Diego Naval Base since 7-16-72, and is a cook.

Cook has rank of captain, has been stationed on the Enterprise since 4-15-76, and is the chief navigator.

```
(ship s_name: 'Enterprise' s_shipclas: 'aircraft_carrier'
  s_captain: 'Perry')
(shipclass $sc_class: 'aircraft_carrier' sc_name: 'Enterprise')
(person p_name: 'Perry' p_rank: 'commodore' p_jobclass: 'captain'
  p_platform: 'Enterprise' p_date_assigned: '710606')
(person p_name: 'Fisher' p_rank: 'seaman' p_jobclass: 'clerk'
  p_platform: 'San Diego Naval Base' p_date_assigned: '741002')
(person p_name: 'Cooper' p_rank: 'seaman' p_jobclass: 'cook'
  p_platform: 'San Diego Naval Base' p_date_assigned: '720716')
(person p_name: 'Cook' p_rank: 'captain' p_jobclass: 'chief_navigator'
  p_platform: 'Enterprise' p_date_assigned: '760415')
```

```
USER B - OPEN %TOP.navyfile.person WRITE;
        UPDATE ship WITH name EQ 'Enterprise'
          captain='Cook';
        UPDATE person WITH name EQ 'Cook'
          duty='captain';
          date_assigned='780921';
```

COPE - The ship with name 'Enterprise' has 'Perry' assigned as captain. Please reassign 'Perry'.

```
USER B - UPDATE person WITH name EQ 'Perry'
          platform=NULL;
          duty=NULL;
          status='inactive';
          date_assigned='780921';
```

```
USER B - UPDATE person WITH name EQ 'Fisher'
          platform='Enterprise';
          jobclass='chief_navigator';
          date_assigned='780921';
```


COPE - The person with name 'Fisher' has rank 'seaman' and cannot be assigned as 'chief_navigator'.
Please specify disposition: delete, replace, need help?

USER B - HELP

COPE - Rule I4: A person whose job is 'chief_navigator' must have rank of at least ensign.

USER B - UPDATE person WITH name EQ 'fisher'
rank = 'ensign';

The user's attempt to change 'Cook' to 'captain' of the 'Enterprise' is converted into ICMDL as:

```
(ship s_name: 'Enterprise' (:= s_captain: 'Cook'))  
(person p_name: 'Cook' (:= p_jobclass: 'captain')  
  (:= p_date_assigned: '780921')).
```

The template for the relation 'ship' is referenced and since the argument 'captain' is being updated, the constraint rule list applying to that argument, (A1 I1), is selected. The rule list for the jobclass argument of the 'person' template, (A4 I2 I4), is also added to the context.

Rules A1 and A4 check for semantic class errors, of which there are none in these updates. The rule I1 states that a ship has exactly one captain:

```
(AND (FOREACH x) (EQUAL (COUNT (SET_UNION y z)) 1)  
  (ship $s_name: (s_name = x) $s_captain: (:= s_captain z))).
```

However, when this is checked for x = 'Enterprise', it retrieves the tuple:

```
(ship s_name: 'Enterprise' s_captain: 'Perry').
```

COPE therefore informs the user of the error, namely that there are two captains specified for the Enterprise. It asks the user if he wants to reassign 'Perry'. User B responds by updating the person relation for 'Perry':

```
(person p_name: 'Perry' (:= p_platform: NULL) (:= p_duty: NULL)  
  (:= p_status: 'inactive') (:= p_date_assigned: '780921')).
```

This is acceptable to COPE and the previous update for 'captain' is allowed. The rules I2 and I4 are not violated, so the update is accepted.

The user's update to change Fisher's assignment is translated:

```
(person p_name: 'Fisher' (:= p_platform: 'Enterprise')  
  (:= p_jobclass: 'chief_navigator') (:= p_date_assigned: '780921')).
```

The rules selected by the person template are (A4 I2 I3) for platform, (A4 I2 I4) for jobclass, and (A4 I2 I5 I13) for date_assigned. It is found that rule I4:

```
(AND (FOREACH x) (GTEQ y 'ensign')
      (person $p_name: (p_name = x) $p_rank: (:= p_rank y)
        $p_jobclass: (p_jobclass = 'chief_navigator'))))
```

is violated since Fisher's rank is 'seaman' which is lower than 'ensign'. This error is communicated to the user who asks for more help in determining the problem. COPE responds by stating the rule and the user is then able to decide what to change.

User B completes his updates relating to this context and the state of the DBMS is now as follows:

Commodore Perry is in an inactive status.
Cook is captain of the Enterprise.
Fisher has rank of ensign, is stationed on the Enterprise, and is the chief navigator.
The date_assigned for these updates is '780921'.

```
(person p_name: 'cook' p_rank: 'captain' p_duty: 'captain'
  p_platform: 'Enterprise' p_date_assigned: '780921')
(person p_name: 'Perry' p_rank: 'commodore' p_duty: NULL
  p_platform: NULL status: 'inactive' p_date_assigned: '780921')
(person p_name: 'Fisher' p_rank: 'ensign' p_duty: 'chief_navigator'
  p_platform: 'Enterprise' p_date_assigned: '780921')
```

* USER C now attempts his updates *

USER C - UPDATE person WITH name EQ 'Perry'
platform='San Diego Naval Base';
date_assigned='780921';

COPE - The person with name 'Perry' has status 'inactive' and
cannot be assigned to a platform.
Please specify disposition: delete, replace, need help?

USER C - DELETE

USER C - UPDATE person WITH name EQ 'Cooper'
platform='Enterprise';
rank='ensign';
date_assigned='780921';
UPDATE person WITH name EQ 'Fisher'
platform='Enterprise';
rank='ensign';
date_assigned='780921';

COPE - The person with name 'Fisher' was assigned on '780921' and
 cannot be reassigned for 30 days.
 Please specify disposition: delete, replace, need help?

USER C - ABORT

In this example, the update:

```
(person p_name: 'Perry' (:= p_platform: 'San_Diego_Naval_Base')  
          (:= p_date_assigned: '780921'))
```

requires checking of rules (A4 I2 I3 I5 I13) as for User B's updates. Rule I2:

```
(person $p_name: (p_name = x) $p_status: (p_status = 'active'))
```

which specifies that a person must be in an active status in order to have his platform or date_assigned changed is violated. COPE informs User C who chooses to delete the update request.

The update:

```
(person p_name: 'Fisher' (:= p_platform: 'Enterprise')  
          (:= p_rank: 'ensign') (:= p_date_assigned: '780921'))
```

violates rule I5 which states that a person cannot be reassigned within 30 days. When notified of this error, User C decides to abort the context. Note that if User C's updates had occurred before User B's, they would have been verified.

Scenario showing user override

Since the data base expert is not infallible, the user is allowed, in some cases, to override COPE.

Current data base state:

The Kennedy is an aircraft carrier.

Perry has rank of commodore and is on inactive status.

(ship s_name: 'Kennedy' s_shipclas: 'aircraft_carrier')

(person p_name: 'Perry' p_rank: 'commodore' p_status: 'inactive')

USER B - UPDATE person WITH name EQ 'Perry'
 platform='Kennedy'
 duty='captain';
 UPDATE ship WITH name EQ 'Kennedy'
 captain='Perry';

COPE - The person with name 'Perry' has status 'inactive' and
 cannot be assigned to a platform.
 Please specify disposition: delete, replace, need help?

USER B - (* hits special key for override *)

COPE - accepts the update, but flags it so it will be known that there
 may be an error or inconsistency in the data base.

The updates:

(person p_name: 'Perry' (:= p_platform: 'Kennedy') (:= p_duty: 'captain'))
(ship s_name: 'Kennedy' (:= s_captain: 'Perry'))

are in violation of rule I2, assigning a person with inactive status, as in the previous scenario. However, in this case the user elects to ignore COPE's warning and to perform the update. COPE passes the update on to the DBMS but attempts to record the inconsistency for future reference.

Scenario using backtracking

User A assigns the Kennedy to Pacific Zone 2 and updates the statistics such as speed, time of arrival, etc. However, the update request is suspended because the ship has insufficient fuel to reach the destination by the expected time of arrival.

In the meantime, User B assigns the Enterprise to Pacific Zone 2 and that request is accepted. When the access list is checked, it is found that backtracking must be done for User A to undo assigning of the Kennedy.

Current data base state:

The Kennedy is an aircraft carrier.

The Kennedy is assigned to Zone 1.

At 8:30 on July 23, 1978, the Kennedy's fuel amount was 180 gallons.

The Enterprise is an aircraft carrier.

The Enterprise is assigned to Zone 1.

Statistics for aircraft carriers include: maximum speed = 35 knots, fuel capacity = 40000 gallons, gallons per hour = 45.

```
(ship s_name: 'Kennedy' s_assignment: 'Pacific_Zone_1' s_fuel_amt: 180
  s_shipclas: 'aircraft_carrier' s_time: '7808230830')
```

```
(ship s_name: 'Enterprise' s_assignment: 'Pacific_Zone_1' s_fuel_amt: 2630
  s_shipclas: 'aircraft_carrier')
```

```
(shipclass $sc_class: 'aircraft_carrier' sc_maxspeed: 35 sc_fuel_cap: 40000
  sc_galls_per_hr: 45 sc_name: (SET := ('Kennedy' 'Enterprise')))
```

```
USER A  - UPDATE ship WITH name EQ 'Kennedy'
          assignment='Pacific_Zone_2'
          heading= *to pacific zone 2*
          speed=30
          dest_lat='6000N'
          dest_long='3000W'
          eta='7808271420'
```

Of particular interest in User A's request are the updates to the arguments 'speed', 'assignment', and 'eta'. Rule 17 states that a ship's speed cannot exceed the maximum for that type:

```
(AND (LTEQ y w)
  (ship $s_name: (s_name = x) $s_shipclas: (:= s_shipclas z))
  (shipclass $sc_class: (sc_class = z) $sc_maxspeed:
    (:= sc_maxspeed w))))
```

Rule 16 requires that if the Kennedy is moved from Pacific Zone 1, another aircraft carrier must be assigned to that zone:

```
(OR
  (ship $s_name: (s_name = x) $s_assignment: (s_assignment = y))
  (AND
    (ship $s_name: (s_name = x) $s_assignment: (:= s_assignment z))
    (ship $s_name: (s_name = [x]) $s_shipclas:
      (s_shipclas = 'aircraft_carrier') $s_assignment:
      (s_assignment = z))))))
```

Both of these rules are verified. Rule I10 specifies a calculation for the necessary fuel amount:

```
(AND (GTEQ v (MULTIPLY (SUBTR y z) w))
  (ship $s_name: (s_name = x) $s_shipclas: (:= s_shipclas t)
    $s_fuel_amt: (:= s_fuel_amt v) $s_eta: (:= s_eta y)
    $s_time: (:= s_time z))
  (shipclass $sc_class: (sc_class = t) $sc_galls_per_hr:
    (:= sc_galls_per_hr w))))
```

However, this check fails because there is insufficient fuel. The violation condition is saved, the M/R task is notified, and the VERIFY task is suspended for this context.

Assume that before User A is able to correct the problem (for instance, by scheduling a tanker to refuel the Kennedy), User B posts a similar request:

```
USER B - UPDATE ship WITH name EQ 'Enterprise'
        assignment='Pacific_Zone_2'
        heading= * to pacific zone 2 *
        speed=32
        dest_lat='5500N'
        dest_long='3200W'
        eta='7808271600';
```

COPE performs the same error checks as for User A. In this case, no errors are found. In particular, rule I10 is successful since the Enterprise has ample fuel. Since User B has nothing further to add the context is closed and the LOCAL DATA task passes the updates to the DBMS. LOCAL DATA also checks the access list for the modified data and flags the BACKTRACK task to reverify User A's context. BACKTRACK discovers a contradiction since the assignment of the Enterprise to Pacific_Zone_1 is a fact used in the proof tree. In this example, a data base reference fails to find any other aircraft carrier in Pacific_Zone_1. Therefore rule I6 no longer succeeds. The M/R task is notified of the error and the context is suspended.

If this example had included another aircraft carrier in Pacific_Zone_1, for instance:

```
(ship s_name: 'Constellation' s_assignment: 'Pacific_Zone_1'
  s_shipclas: 'aircraft_carrier')
```

then BACKTRACK could have substituted this fact for the changed item in User A's context. BACKTRACK would have completed successfully and passed the reverified context on to the VERIFY task.

8. REFERENCES

1. Aho, A.V., and Ullman, J.D.; The Theory of Parsing, Translation, and Compiling, volume 1: Parsing; Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.
2. Allman, E. and Stonebraker, M.; "Embedding a Relational Data Sublanguage in a General Purpose Programming Language"; SIGPLAN Notices, Special Issue, March 1976, pp. 25-35.
3. Brandenburg, R.; Advanced Command Control Architectural Testbed System Architecture and Capabilities FY 1977; Systems Exploration, Inc., San Diego, California, April 1977.
4. Brown, G.; Expanded Relational Models for the Fleet Command Center and At-Sea Commander's Databases; Systems Exploration, Inc., San Diego, California, March 1977.
5. Brown, G.; The Relational Model for the Bluefile Data Base; Systems Exploration, Inc., San Diego, California, November 1976.
6. Canaday, R.H., Harrison, R.D., Ivie, E.L., Ryder, J.L., and Wehr, L.A.; "A Backend Computer for Data Base Management"; Communications of the ACM; Vol. 17, No. 10 (October 1974), pp. 575-582.
7. Chamberlin, D.D., Astrahan, M.M., Eswaran, K.P., Griffiths, P.P., Lorie, R.A., Mehl, J.W., Reisner, P., and Wade, B.W.; "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control"; IBM Journal of Research and Development, Vol. 20, No. 6 (November 1976), pp. 560-575.
8. "CODASYL Data Description Language"; Journal of Development, Handbook 113, National Bureau of Standards, Washington, D.C., 1974.
9. Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus"; Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego, California, November 11-12, 1971.
10. Datacomputer Version I User Manual; Computer Corporation of America, Cambridge, Massachusetts, August 1975.
11. Date, C.J.; An Introduction to Database Systems; Addison-Wesley, Reading, Massachusetts, 1975.
12. Davis, R., and King, J.; "An Overview of Production Systems"; Machine Representations of Knowledge, Proceedings of NATO Advanced Study Institute, Santa Cruz, California (June 1975), in preparation (also available as AIM 271, Stanford University, Stanford, California).

13. Duda, R.O., Hart, P.E., Nilsson, N.J., Reboh, R., Slocum, J., and Sutherland, G.L.; Development of a Computer-Based Consultant for Mineral Exploration; SRI International, October 1977.
14. Eswaran, Kapali P. and Chamberlin, Donald D.; "Functional Specifications of a Subsystem for Data Base Integrity"; Proceedings of the International Conference of Very Large Data Bases (September 22-24, 1975), Association of Computing Machinery, New York, 1975, pp. 48-68.
15. Fernandez, E.B. and Summers, R.C.; "Integrity Aspects of a Shared Data Base"; Proceedings of the National Computer Conference 1976, American Federation of Information Processing, New York, 1976, pp. 204-1 - 204-9.
16. Florentin, J.J.; "Consistency Auditing of Data Bases"; Computer Journal, vol. 17, number 1 (February 1974), pp. 52-58.
17. Floyd, R.W.; "Assigning Meanings to Programs"; Proceedings of the Symposium of Applied Mathematics, American Mathematics Society, vol. 19, 1967, pp. 19-32.
18. GACT Multi-terminal User's Manual; NCCS Document PME-108-M00012, Naval Electronic Systems Command, PME-108, Washington, D.C., August 1976.
19. Graves, R.W.; "Integrity Control in a Relational Data Description Language"; Proceedings of the ACM Conference '75 Pacific, Association of Computing Machinery, New York, 1975, pp. 108-113.
20. Hammer, M.M. and McLeod, D.J.; "Semantic Integrity in a Relational Data Base System"; Proceedings of the ACM-RAND International Symposium on Very Large Data Bases, Boston, 1975, pp. 25-47.
21. Hendrix, G.G.; "Expanding the Utility of Semantic Networks Through Partitioning"; Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tblisi, Georgia, USSR (1975), pp. 115-121.
22. Hoare, C.A.R.; "An Axiomatic Basis for Computer Programming"; Communications of the ACM, Vol. 12, No. 10 (October 1969), pp. 576-580.
23. Hoare, C.A.R.; "Procedures and Parameters: An Axiomatic Approach"; in Symposium on Semantics of Algorithmic Languages, E. Engeler (Ed.) Springer-Verlag, 1971, pp. 102-116.
24. Klass, P.J.; "ARPA Net Aids Command, Control Tests"; Aviation Week and Space Technology, McGraw Hill, Inc., vol. 102, no. 38 (September 27, 1976), pp. 63-67.
25. McLeod, D. and Meldman, M.; "RISS - A Generalized Minicomputer Relational Data Base Management System"; Proceedings of the National Computer Conference, Vol. 44, American Federation of Information Processing, 1975, pp. 397-402.

26. McSkimin, J.; The Use of Semantic Information in Deductive Question-Answering Systems; Ph.D. Dissertation, Department of Computer Science, University of Maryland, 1976.
27. McSkimin, J. and Minker, J.; The Use of a Semantic Network in a Deductive Question-Answering System; Technical Report TR-506, Department of Computer Science, University of Maryland, 1977.
- 28.
29. Quillian, M.R.; "Semantic Memory"; in Minsky, M. (ed.) Semantic Information Processing, MIT Press, Cambridge, Mass., 1969.
30. Reboh, R., Raphael, B., Yates, R.A., Kling, R.E., and Verlarde, C.; Study of Automatic Theorem-Proving Programs; Stanford Research Institute, Menlo Park, California, Technical Note 73, 1972.
31. Schneider, G.M., Weingart, S.W., and Perlman, D.M.; An Introduction to Programming and Problem Solving with PASCAL; John Wiley and Sons, New York, 1978.
32. Shortliffe, E.H.; Computer-Based Medical Consultations: MYCIN; American Elsevier Publishing Company, New York, 1976.
33. Small, D.L. and Christy, D.O.; Command Center Information System (CCIS) Functions and Capabilities; Technical Document 498, Naval Electronics Laboratory Center, San Diego, California, 1976.
34. Stonebraker, M.; "Implementation of Integrity Constraints and Views by Query Modification"; Proceedings 1975 SIGMOD Workshop on Management of Data, San Jose, California, May 1975.
35. Stonebraker, M., Wong, E., Kreps, P., and Held, G.; The Design and Implementation of INGRES; Memorandum No. ERL-M577, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, January 1976.
36. Wilson, G.A.; A Description and Analysis of the PAR Technique -- An Approach to Parallel Search and Parallel Inference in Problem Solving Systems; Ph.D. Thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 1976.
37. Volpe, J.R. and Brandenburg, R.L.; A Relational Model for a Fleet Command Center Data Base; Technical Note 3206, Naval Electronics Laboratory Center, San Diego, California, August 1976.
38. Zloof, M.M.; Security and Integrity within the Query-by-Example Data Base Management Language; Research Report RD 6982, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, February 1978.

APPENDIX A - INTERNAL CONCEPTUAL MODEL DESCRIPTION LANGUAGE (ICMDL) SYNTAX

TOP LEVEL

```

[COPE_MODEL] ::=      ( [CONCEPTUAL_FRAMEWORK] [DATA_STMT_LIST] )
[CONCEPTUAL_FRAMEWORK] ::=
                        ( [DB_STRUCTURE_LIST] [ISA_LIST] [TEMPLATE_LIST]
                          [CONSTRAINT_RULE_LIST] )

```

DATA BASE STRUCTURE - DATA TYPE DECLARATIONS

```

[DB_STRUCTURE_LIST] ::=
                        [DB_STRUCTURE_LIST] [DB_STRUCTURE] !
                        [DB_STRUCTURE]
[DB_STRUCTURE] ::=      ( FILE [FILE_NAME] [TYPE] )
[TYPE_DECL_LIST] ::=    [TYPE_DECL_LIST] [TYPE_DECL] !
                        [TYPE_DECL]
[TYPE_DECL] ::=         ( TYPE [IDENTIFIER] [TYPE] )
[TYPE] ::=              [STRUCTURED_TYPE] !
                        [TYPE_IDENTIFIER]
[TYPE_IDENTIFIER] ::=   BOOLEAN !
                        REAL !
                        INTEGER !
                        CHARACTER !
                        TEXT !
                        [IDENTIFIER]
[STRUCTURED_TYPE] ::=   [ARRAY_TYPE] !
                        [SET_TYPE] !
                        [BAG_TYPE] !
                        [LIST_TYPE] !
                        [RECORD_TYPE]

```

OTHER STRUCTURED TYPES

```

[ARRAY_TYPE] ::=      ( ARRAY [ARRAY_DIMENSION] OF ( [TYPE] ) )
[ARRAY_DIMENSION] ::= ( [INTEGER] [INTEGER] )
[SBLR_ELEMENT] ::=     [TYPE_IDENTIFIER] !
                        [TYPE_DECL]
[SET_TYPE] ::=         ( SET WITH SIZE = [SIZE] OF [SBLR_ELEMENT] )
[BAG_TYPE] ::=         ( BAG WITH SIZE = [SIZE] OF [SBLR_ELEMENT] )
[LIST_TYPE] ::=        ( LIST WITH TERMINATOR = [CHAR] OF
                        [SBLR_ELEMENT] ) !
[SIZE] ::=             ( LIST WITH SIZE = [SIZE] OF [SBLR_ELEMENT] )
                        [INTEGER] !
                        [DB_FIELD_NAME]

```

RECORD TYPE

```

[RECORD_TYPE] ::= ( RECORD ( [RECORD_FIELD_LIST] ) )
[RECORD_FIELD_LIST] ::=
    [RECORD_FIELD_LIST] [RECORD_FIELD] !
    [RECORD_FIELD]
[RECORD_FIELD] ::= ( [RECORD_FIELD_NAME] : [SBLR_ELEMENT] ) !
    [CASE_PART]
[RECORD_FIELD_NAME] ::= [IDENTIFIER]
[CASE_PART] ::= ( CASE [SELECTOR] ( [CASE_BLOCK] ) )
[CASE_BLOCK] ::= [CASE_BLOCK] ( ( [CASE_LABEL_LIST] )
    ( [RECORD_FIELD_LIST] ) ) !
    ( ( [CASE_LABEL_LIST] ) ( [RECORD_FIELD_LIST] ) )
[CASE_LABEL_LIST] ::= [CONSTANT_LIST]
[SELECTOR] ::= [DB_FIELD_NAME]
[CONSTANT_LIST] ::= [CONSTANT_LIST] [CONSTANT] !
    [CONSTANT]
[DB_FIELD_NAME] ::= [DB_FIELD_PATHNAME] !
    [DB_FIELD_PATHNAME] [ARRAY_INDEX]
[DB_FIELD_PATHNAME] ::= [DB_FIELD_PATHNAME] . [IDENTIFIER] !
    [IDENTIFIER]
[ARRAY_INDEX] ::= [ARRAY_INDEX] [INTEGER_EXPRESSION] !
    [INTEGER_EXPRESSION]
[INTEGER_EXPRESSION] ::=
    ( [ARITH_OPERATOR] ( [INTEGER_EXPRESS_LIST] ) ) !
    [INTEGER] !
    - [INTEGER] !
    [FORM] !
    [TERM]
[INTEGER_EXPRESS_LIST] ::=
    [INTEGER_EXPRESS_LIST] [INTEGER_EXPRESSION] !
    [INTEGER_EXPRESSION]
[ARITH_OPERATOR] ::= [IDENTIFIER]

```

ISA HIERARCHY

```

[ISA_LIST] ::= [ISA_LIST] [ISA_ELEMENT] !
    [ISA_ELEMENT]
[ISA_ELEMENT] ::= ( ISA [ISA_IDENTIFIER_NAME] [COLLECTION] )
[COLLECTION] ::= [LOCAL_COLLECTION] !
    ( [OPERATOR] [COLLECTION] [COLLECTION] )
[LOCAL_COLLECTION] ::= [ISA_IDENTIFIER] !
    ( SET := [SBL_DESIG] ) !
    ( BAG := [SBL_DESIG] ) !
    ( LIST := [SBL_DESIG] ) !
    ( TEXT := [CHAR_STRING] )
[CHAR_STRING] ::= [CHAR_STRING] [CHAR] !
    [CHAR]
[SBL_DESIG] ::= ( [CONSTANT_LIST] ) !
    [DB_DESIGNATOR]

```

```

[DB_DESIGNATOR] ::=      ( SELECT ( [FIELD_LIST] ) ) !
                        ( PROJECT ( [FIELD_LIST] ) ) !
                        ( JOIN ( [JOIN_LIST] ) )
[JOIN_LIST] ::=          [JOIN_LIST] ( [DB_DESIGNATOR] ) !
                        ( [DB_DESIGNATOR] ) ( [DB_DESIGNATOR] )
[FIELD_LIST] ::=         [FIELD_LIST] [FIELD] !
                        [FIELD]
[FIELD] ::=              ( [RELATION_NAME] [ISA_TERM_LIST] )
[ISA_TERM_LIST] ::=      [ISA_TERM_LIST] [ISA_TERM] !
                        [ISA_TERM]
[ISA_TERM] ::=           ( [ARG_NAME] = [ISA_RESTRICT] ) !
                        ( := [ARG_NAME] )
[ISA_RESTRICT] ::=       [PRIME_OBJ] !
                        [ [PRIME_OBJ] ]
[PRIME_OBJ] ::=          [CONSTANT] !
                        [LOCAL_COLLECTION]
[ISA_IDENTIFIER_NAME] ::=
                        [IDENTIFIER]
[ISA_IDENTIFIER] ::=     [IDENTIFIER]
*****

```

OPERATORS

```

-----
[OPERATOR] ::=           [SET_OPERATOR] !
                        [BAG_OPERATOR] !
                        [LIST_OPERATOR]
[SET_OPERATOR] ::=       SET_UNION !
                        SET_INTERSECTION !
                        SET_DIFFERENCE
[BAG_OPERATOR] ::=       BAG_INTERSECTION !
                        BAG_ADD !
                        BAG_DIFFERENCE
[LIST_OPERATOR] ::=      LIST_APPEND
*****

```

TEMPLATE

```

-----
[TEMPLATE_LIST] ::=      [TEMPLATE_LIST] [TEMPLATE] !
                        [TEMPLATE]
[TEMPLATE] ::=           ( RELATION [RELATION_NAME] TEMPLATE ( [PARM_LIST] ) )
[PARM_LIST] ::=          [PARM_LIST] [PARM] !
                        [PARM]
[PARM] ::=               ( [ARG_NAME] :: [SUBFIELD] )
[SUBFIELD] ::=           [DB_FIELD_NAME] : ( [RULE_NAME_LIST] ) !
                        ( ARRAY [ARRAY_DIMENSION] OF [SUBFIELD] ) !
                        ( SET OF [SUBFIELD] ) !
                        ( BAG OF [SUBFIELD] ) !
                        ( LIST OF [SUBFIELD] ) !
                        [T_RECORD]
[RULE_NAME_LIST] ::=     [RULE_NAME_LIST] [RULE_NAME] !
                        [RULE_NAME]

```

```

[T_RECORD] ::= ( RECORD ( [T_RECORD_FLD_LST] ) )
[T_RECORD_FLD_LST] ::= [T_RECORD_FLD_LST] [T_RECORD_FLD] !
[T_RECORD_FLD] ::= ( [RECORD_FIELD_NAME] :: [SUBFIELD] ) !
[T_CASE_PART] ::= ( CASE [SELECTOR] ( [T_CASE_BLOCK] ) )
[T_CASE_BLOCK] ::= [T_CASE_BLOCK] ( [CASE_LABEL_LIST] )
( [T_RECORD_FLD_LST] ) !
( [CASE_LABEL_LIST] ) ( [T_RECORD_FLD_LST] )
*****

```

RULES

```

[CONSTRAINT_RULE_LIST] ::=
[CONSTRAINT_RULE_LIST] [RULE] !
[RULE] ::=
( RULE [RULE_NAME] [PATTERN] [FORMULA] [ACTION]
[COST] [PAYOFF] )
[RULE_NAME] ::= [IDENTIFIER]
[COST] ::= [NUMERIC]
[PAYOFF] ::= [NUMERIC]
[PATTERN] ::= [ATOM2]
[ACTION] ::= [ACTION_LIST]
[ACTION_LIST] ::= [ACTION_LIST] [IDENTIFIER] !
[IDENTIFIER]
[FORMULA] ::=
(( [TYPE_DECL_LIST] )( [INTERIOR_FORM_LIST] )) !
( ( [INTERIOR_FORM_LIST] ) )
[INTERIOR_FORM_LIST] ::=
[INTERIOR_FORM_LIST] [INTERIOR_FORM] !
[INTERIOR_FORM] ::=
( [SIMPLE_FORM] ) !
[FORMULA]
*****

```

SIMPLE FORMULA

```

[SIMPLE_FORM] ::= [LITERAL] !
( [CONNECTIVE] [SIMPLE_FORM] ) !
( IMPLIES [SIMPLE_FORM] [SIMPLE_FORM] )
[LITERAL] ::= [ATOM] !
NOT [ATOM]
[ATOM] ::= [ATOM1] !
[ATOM2]
[ATOM1] ::= ( [EX_PREDICATE] [EX_TERM_LIST] )
[EX_TERM_LIST] ::= [EX_TERM_LIST] [EX_TERM] !
[EX_TERM]
[EX_TERM] ::= [PRIMARY_OBJECT] !
( [EX_FORM] )
[EX_FORM] ::= [FUNCTION] [EX_TERM_LIST] !
:= [EX_TERM] [VARIABLE]

```

```

[PRIMARY_OBJECT] ::= [CONSTANT] !
                    [LOCAL COLLECTION] !
                    [VARIABLE]
[ATOM2] ::= ( [RELATION_NAME] [REL_TERM_LIST] )
[REL_TERM_LIST] ::= [REL_TERM_LIST] [REL_TERM] !
                    [REL_TERM]
[REL_TERM] ::= $[ARG_NAME]: [PRIMARY_TERM]
[ARG_NAME] ::= [IDENTIFIER]
[PRIMARY_TERM] ::= [ARG_NAME] !
                  ( [FORM] ) !
                  ( [ARG_NAME] = [RESTRICTION] )
[FORM] ::= [FUNCTION] [PRIMARY_TERM_LIST] !
           := [PRIMARY_TERM] [VARIABLE]
[PRIMARY_TERM_LIST] ::= [PRIMARY_TERM_LIST] [PRIMARY_TERM] !
                       [PRIMARY_TERM]
[RESTRICTION] ::= [PRIMARY_OBJECT] !
                  [ [PRIMARY_OBJECT] ]

```

MISCELLANEOUS NON-TERMINALS

```

[EX_PREDICATE] ::= [IDENTIFIER]
[FILE_NAME] ::= [IDENTIFIER]
[RELATION_NAME] ::= [IDENTIFIER]
[VARIABLE] ::= [IDENTIFIER]
[FUNCTION] ::= [IDENTIFIER]
[CONSTANT] ::= '[CHAR_STRING]'

```

DATA STATEMENTS

```

[DATA_STMT_LIST] ::= [DATA_STMT_LIST] [DATA_STMT] !
                    [DATA_STMT]
[DATA_STMT] ::= ( [RELATION_NAME] [DATA_TERM_LIST] )
[DATA_TERM_LIST] ::= [DATA_TERM_LIST] [DATA_TERM] !
                    [DATA_TERM]
[DATA_TERM] ::= [ARG_NAME] : [PRIME_OBJ] !
                (:= [ARG_NAME] : [PRIME_OBJ] )

```

APPENDIX B - INTERNAL CONCEPTUAL MODEL DESCRIPTION LANGUAGE (ICMDL) SEMANTICS

This section describes the semantic constraints of ICMDL.

- * The COPE model is the top level of the ICMDL syntax and contains the conceptual framework and the data statements.
- * The conceptual model consists of four basic components:
 - 1) [DB_STRUCTURE_LIST] - the data base structure description
 - 2) [ISA_LIST] - the semantic dictionary
 - 3) [TEMPLATE_LIST] - the relation templates
 - 4) [CONSTRAINT_RULE_LIST] - the constraint rules

** The Data Base Structure Description **

- * The structural description of the data base is represented in a file format. There is a one-to-one and onto mapping between the template relations and the FILE declarations of the data base structure description.
- * The statement [TYPE_DECL] ::= (TYPE [IDENTIFIER] [TYPE]) gives the ability to define and name a structure that can be used repeatedly to define other data base structures. Since the [TYPE] may be a complex structure type or may refer to one defined elsewhere, the capability for hierarchical structures is provided.
- * Arrays may only be one-dimensional; however, the effect of multiple dimensions can be achieved by using nesting and declaring the elements to be arrays.
- * The elements of arrays, sets, bags, lists, and record fields may be any type except FILE. All elements of arrays, sets, bags, and lists must be of the same type. Only the RECORD type allows fields of different types.
- * The cardinality of lists, sets, and bags must be expressed by stating the size, which may be contained in a pointer. For lists, an alternative is to specify a termination character.
- * Lists are ordered; sets and bags are not.
- * The RECORD type specifies a structure with fields that may be of any type except FILE. The CASE statement allows the definition of records which

are versions of a common record. Record fields can be conditionally defined, depending on the value of a [SELECTOR]. The selector is a previously defined [DB_FIELD_NAME] which need not be a field of the same record. Note that it is the fields themselves which are conditionally created, not the contents of the field. This approach was taken to preserve strong typing. Each case block contains at least one [CASE_LABEL_LIST] which states the values that the selector can take on for the corresponding [RECORD_FIELD_LIST] to be declared. The elements of the [CASE_LABEL_LIST] must be of the same type as the selector and the value of the selector must appear in some [CASE_LABEL_LIST].

- * The pathname is a unique name formed as the concatenation of the names of all the parent structures up through the file name. The general form for expressing it is:

name1 . name2 namen

where the outermost levels occur first. As an example of a pathname, consider the data base structure:

```
(FILE ship (LIST WITH TERMINATOR = EOF OF
  (TYPE ship_struct (RECORD
    ((name: (TYPE name (ARRAY (1 20) OF CHAR)))
    (readings_count: integer)
    (readings: (LIST WITH SIZE = readings_count OF
      (TYPE stats (RECORD
        ((fuel_amt: integer)
        (speed: integer))))))
    (captain: name))))))
```

The complete pathname to reference "speed" is:
ship.ship_struct.readings.stat.speed

- * The [DB_FIELD_NAME] consists of a pathname and may also include an array index if the indicated field is an array. An array index may be a computable expression.

** The Semantic Dictionary **

- * An ISA statement: [ISA_ELEMENT] ::= (ISA [ISA_IDENTIFIER_NAME] [COLLECTION]) specifies the classes to which each argument of each relation belongs. [ISA_IDENTIFIER_NAME] is a logical name for referring to the declaration, the semantic class name. [COLLECTION] specifies the type which may be:

- 1) the explicit specification of a set, bag, list, or text
- 2) a previously defined identifier, usually representing a complex structure
- 3) a data base designation which includes any of the relational operators SELECT, PROJECT, or JOIN.

- * An ISA statement can return only one argument of the tuples retrieved. That argument may be a list, bag, or set, as previously defined. The left arrow (:=) indicates the argument returned. Any other arguments mentioned are retrieved but not returned.
- * A restriction may be placed on an argument of a relation by using the construction: ([ARG_NAME] = [ISA_RESTRICT]). This allows the specification of conditions for selecting certain data base fields. A restriction may be a constant, identifier, or an explicit set, bag, list, or text but must agree in type with the argument. An example of a restriction is presented in the following line which specifies that a captain is a person with rank of "captain":

```
(ISA scaptain (SET := (PROJECT
    (person (:= p_name) (p_rank = 'captain')))))
```

The argument "p_rank" is restricted to having the value "captain".

- * Brackets around a restriction indicate the complement of the enclosed element.
- * The purpose of the functions SELECT, PROJECT, and JOIN is to state what data base retrievals are necessary to create the explicit list, bag, or set which corresponds to the semantic category being defined. Each returns a collection of tuples, the fields of which are determined by the restrictions. SELECT returns tuples containing all fields of the relation, while PROJECT returns tuples with only those fields that were explicitly requested. JOIN retrieves tuples which may be a combination of fields from different relations.
- * In the construction ([OPERATOR] [COLLECTION] [COLLECTION]), the types of all three components must agree (ie. all sets or all bags or all lists).

** The Relation Templates **

- * For every template there is a corresponding unique data base structure. The filename given in the structure definition is the name given to the relation in the template. The elements of the parameter list each correspond to a structure in the FILE and the exact correspondence is represented by the logical [ARG_NAME] and the pathname. For example, the relation:

```
((RELATION ship TEMPLATE
    (s_name:: ship_struct.name)))
```

corresponds to the data base structure:

```
(FILE ship (LIST WITH TERMINATOR = EOF OF
    (TYPE ship_struct (RECORD
        ((name: (TYPE name (ARRAY (1 20) OF CHAR)))))))
```

- * The templates show the composition of the structure and indicate at what levels rules should be applied. Rules may be applied to a structure as a whole, to individual elements, or to any other level of the structure. In a replicated structure, a rule defined on a level must be applied to every instance of that level. An example of a template showing rules is:

```
(RELATION person TEMPLATE
  ((s_platform:: person_struct.platform: (I5 I6))))
```

where the rule list (I5 I6) indicates constraints to be applied to an individual element of a relation.

- * Any data base structure defined within a template has a correspondence to a structure in the data base which is pointed to by the pathname. The structure and template must agree in all features such as array dimension, record fields, etc.

** The Constraint Rules **

- * A rule consists of the following parts:

- 1) Name - An identifier which has a twofold purpose:
 - a) It is a means of referencing the rule in case the Data Base Administrator wants to change it.
 - b) It is used in the templates to state which rules are to be invoked when different fields are updated.
- 2) Pattern - The list of variables used in the rule.

Guided by the pattern, the pattern matcher causes the variables of the formula to be instantiated to the values given in the corresponding arguments of the user input. If partial matches are allowed, the elements which are not specified will be considered free variables and defined to default to null or to some universal value. The question of whether this procedure of expanding the user's request should be performed by the translator or the pattern-matcher will be answered at the time of implementation.

The pattern consists of the COPE relation name corresponding to the name of a relation previously defined in a template, followed by a list of relation terms. Each relation term is an argument name, followed by a primary term. The argument name is preceded by a "\$" to indicate that it corresponds to an argument name defined in a template. The purpose of this structure is to remove the position dependency by assignment of the arguments.

- 3) Formula - The predicate-calculus expression of the semantic constraint.

The block-structured language provides efficiency of execution of the semantic constraints and ease of construction of a complex formula by the data base expert. The formula may consist of multiple statements which are tested in a sequential manner, but all must evaluate to be true in order for the constraint to be satisfied. Local variables give efficient and easy reference to sets, bags, or other groups of elements obtained by data base retrievals. These side-effect results can be saved for use in other statements in the list and for possible reference in interactions with the user to provide explanations of reasons for errors. An example of the pattern and formula is given for the constraint "A person can be assigned to at most one platform":

Pattern: (person \$p_name: (:= p_name x)
 \$p_platform: (:= p_platform y))

Formula: (AND (FOREACH x) (LTEQ (COUNT (SET UNION y z)) 1)
 (person \$p_name: (p_name = x) \$p_platform:
 (:= p_platform z)))

If this rule is applied when an update for a change of assignment is received, such as:

(person p_name: 'Smith' (:= p_platform: 'Enterprise'))

then x will be bound to "Smith" and z will contain the set of values of the field person.platform for all tuples with the field person.name equal to "Smith".

- 4) Action - A list of one or more built-in functions within COPE which specify what the system should do when one or more portions of a rule fail. Examples are: LOG (record the error), DELETE (drop this update from the user's context), etc.
- 5) Cost - A real number (0-1.0) which indicates the effort involved in applying a rule.
- 6) Payoff - A real number (0-1.0) which indicates the likelihood that an error will be detected by applying this rule.

* All relations and variables referenced in the rules section must be defined prior to their use in the type declaration section. The references may occur only within the scope of the declaration.

- * The connectives OR and AND are n-ary.
- * A function returns a value; a predicate returns TRUE or FALSE.
- * The values of a variable must agree with the type declared for that variable.
- * The COPE relations defined in the templates are predicates.
- * An assignment of a DB_FIELD_NAME to a variable indicates that all values that are taken on by the DB_FIELD_NAME are being accessed.

**** The Data Statements ****

- * The data statements are the ICMDL representation of the data, both the user update requests and the data returned by the LOCAL DATA process.
- * A data statement consists of the relation name followed by a DATA_TERM_LIST. Each DATA_TERM specifies an argument name and its corresponding value. If the data term includes the assignment operator, the named field is being updated.

APPENDIX C - ALGORITHMS ASSOCIATED WITH THE TASKS

1. Relation Cluster Construction Algorithm

A relation cluster is a connected directed graph where the nodes are relations and the constraint rules specify the paths between nodes. Formally, it is defined in the traditional graph theoretic manner as follows:

Definition: Let CR be a constraint rule and R be a relation. R is part of CR, abbreviated $R \subset CR$, if and only if:

- (a) R appears in the formula portion of CR; or,
- (b) R appears in an [isa_element] which appears in the formula portion of CR.

Definition: Let CR be a constraint rule and T be a template. CR is referenced by T, abbreviated $CR \subset T$, if CR appears in a [rule_list] of T.

Definition: Let $T(R_1)$ be the relation template for relation R_1 , CR be a constraint rule, and R_2 be a relation such that $R_1 \subset R_2$. Relation R_2 is contained in $T(R_1)$, abbreviated $R_2 \subset T(R_1)$, if $R_2 \subset CR$ and $CR \subset T(R_1)$.

Definition: Let R_1 and R_2 be relations such that $R_1 \subset R_2$. A directed arc exists from R_1 to R_2 if $R_2 \subset T(R_1)$.

Definition: Let R_1, R_2, \dots, R_n be n distinct relations. A directed path exists from R_1 to R_n if for each $i, 1 \leq i < n$, a directed arc exists from R_i to R_{i+1} .

Definition: The relation cluster associated with relation R_1 , abbreviated $C(R_1)$, is the set of relations such that:

- (a) R_1 is a member of $C(R_1)$, and
- (b) R_i is a member of $C(R_1)$ if and only if there is a directed path from R_1 to R_i .

The method for constructing the relation cluster associated with relation R_1 is given by the following algorithm:

Algorithm for Constructing a Relation Cluster with Labeling

This algorithm performs the functions of traversing the directed graph associated with relation R_1 , labels R_1 with its descendant set of relations (which is $C(R_1)$), and labels each descendant with its ancestors reachable from R_1 . The sets OPEN and TEMP are used to keep track of the paths emanating from R_1 . It is assumed that the ancestor labels of each relation are initially empty.

```

A1: [Initialize.]

DESC_LABEL(R1) := 0
OPEN, I := R1
GO TO A3

A2: [Traverse an arc.]

IF OPEN = 0 THEN TERMINATE
ELSE
    Remove the next member of OPEN and place it in I
    DESC_LABEL(R1) := DESC_LABEL(R1) U [I]

A3: [Find new arcs.]

TEMP := [R ! R T(I) and R I]
IF TEMP = 0 THEN GO TO A2

A4: [Terminate loops.]

Remove the next member of TEMP and place it in J
IF J ANCES_LABEL(I) THEN GO TO A6

A5: [Establish ancestor label.]

ANCES_LABEL(J) := ANCES_LABEL(I) U [I] U ANCES_LABEL(J)
OPEN := OPEN U [J].

A6: [Test for termination of arc list.]

IF TEMP = 0 THEN GO TO A2
ELSE GO TO A4

```

Because the relation clusters are constructed from directed graphs, the clusters form hierarchies. This enables COPE to maintain the most restrictive view of which relations are dependent on others, and thus maintain contexts at a minimum size. The advantage of keeping the contexts small is that fewer rules need to be applied and therefore the error checks can be completed, and the user input passed on to the DBMS, more quickly.

Figure C-1 shows an illustration of a relation cluster graph. Each node in Figure C-1 is labeled with two sets: the ancestor node set and the descendent node set. The ancestor node set of a node R is the collection of nodes (relations) from which node R may be reached and which are therefore dependent upon R. R is a member of the clusters associated with each of these nodes. The descendent node set is the collection of nodes which may be reached from R, and is the relation cluster for R. For example, the relation R2 is dependent on relations R4, R5, R6, and R7 and is a member of the cluster for R1. These labels are employed by the context determination algorithm to decide whether or not a new user input belongs to a given context.

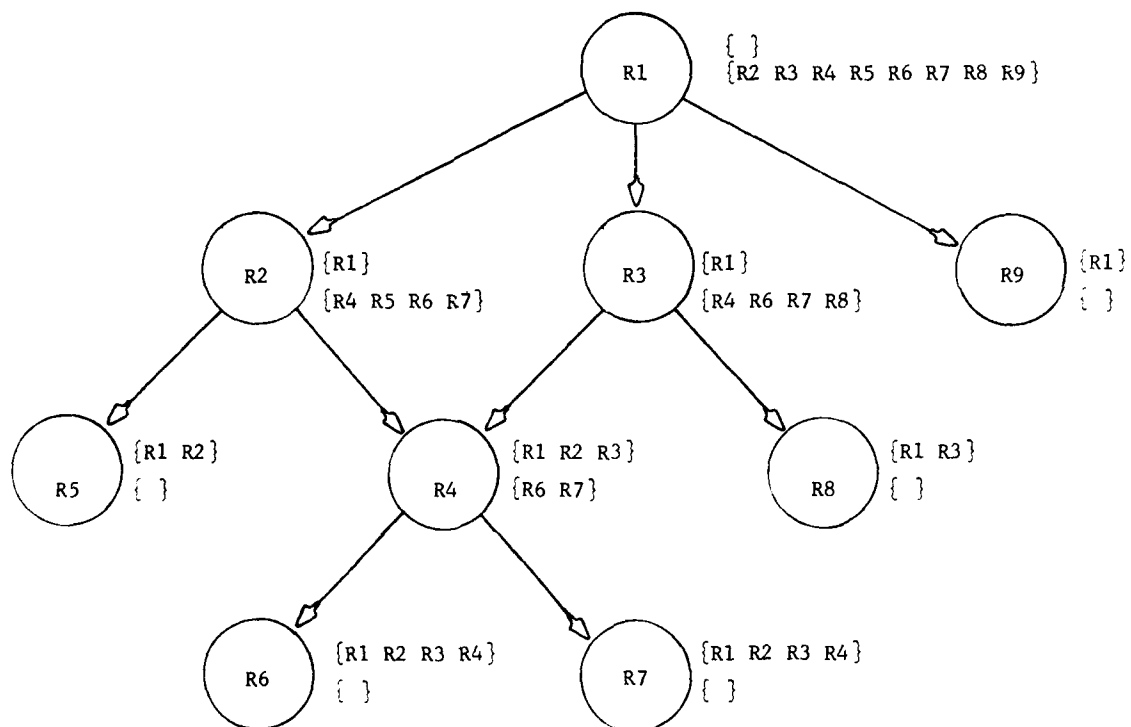


Figure C-1: A Portion of a Relational Cluster Graph

2. Context Determination Algorithm

The formal definition of a context is provided by the following algorithm for context determination:

Algorithm for Context Determination

This algorithm takes the next (or first) user input and adds it to the proper user context. If necessary, it creates a new context for the user.

Let CONTEXT_LIST be a pointer to the list of contexts for a particular user. Each node of this list contains three entries: a pointer to the next node (NEXT), a pointer to the set of user inputs that are members of this context (INPUT) and a pointer to the actual context (CONTEXT).

Let R hold the relation name of the current input. TR is a pointer to the template for relation R. MERGE(X,Y) is a function which merges contexts X and Y by replacing X with the new context and deleting Y.

B1: [Get next input.]

C := CONTEXT_LIST.

B2: [Create new context; check descendants of new input.]

```
IF C = NULL THEN
  I := NEWNODE
  NEXT(I) := CONTEXT_LIST
  CONTEXT_LIST := I
  INPUT(I) := [R]
  TERMINATE
```

[Create a new context and put the entry at the head of CONTEXT_LIST.]

```
ELSE
  M := DESC_LABEL(TR) INPUT(C)
  IF M = 0 THEN GO TO B4
  ELSE
    I := C
```

[If any of the inputs of the context currently being examined are descendants of the relation corresponding to the new input, then save a pointer to the context.]

B3: [Loop, testing descendants and merging contexts.]

```
J := C; C := NEXT(C)
IF C = NULL THEN GO TO B6
M := DESC_LABEL(TR) INPUT(C)
IF M = 0 THEN GO TO B3
ELSE
    CONTEXT(I) := MERGE(CONTEXT(I),CONTEXT(C))
    NEXT(J) := NEXT(C)
    RETURN(C)
    GO TO B3
```

[Compare the descendants of the new input against the inputs of each context. Whenever a match is found, merge that context with the previous matching context (I), replace the first node with the new node, and delete the second node.]

B4: [Check ancestors of new input.]

```
M := ANCES_LABEL(TR) INPUT(C)
IF M = 0 THEN
    C := NEXT(C)
    GO TO B2
ELSE
    I := C
```

[If any of the inputs of the context currently being examined are ancestors of the relation corresponding to the new input, then save a pointer to the context. Otherwise, repeat B2 with the next context.]

B5: [Loop, testing ancestors and merging contexts.]

```
J := C
C := NEXT(C)
IF C = NULL THEN GO TO B6
M := ANCES_LABEL(TR) INPUT(C)
IF M = 0 THEN GO TO B5
ELSE
    CONTEXT(I) := MERGE(CONTEXT(I),CONTEXT(C))
    NEXT(J) := NEXT(C)
    RETURN(C)
    GO TO B5
```

B6: [Terminate.]

```
INPUT(I) := INPUT(I) U [R]
TERMINATE
```

[All contexts have been tested; store the new input and terminate.]

Using the example graph of Figure C-1, Table C-1 shows a trace of the context determination algorithm as it receives a sequence of user inputs. COPE assumes that each user input represents an update to a single relation. For simplicity the example shown in Table C-1 uses only the number to indicate the relation to which the user input refers. Each line of the Table shows the input to be processed next, and the contents of each existing context for this user just prior to the processing of this input. The names for each context are assigned solely for reference in the discussion. The example assumes that none of the user contexts can be closed during the course of this sequence of user inputs. In tracing the example one can see how new contexts are created when different user inputs are not mutually dependent, and how contexts are combined when a new input is entered which is dependent on more than one of the existing contexts.

NEXT INPUT	CONTEXT-A	CONTEXT-B	CONTEXT-C
5	-----	-----	-----
3	5	-----	-----
2	5	3	-----
5	5,2	3	-----
8	5,2,5	3	-----
9	5,2,5	3,8	-----
4	5,2,5	3,8	9
----	5,2,5,3,8,4	9	-----

TABLE C-1: An Example of Context Determination

3. Verification Algorithm

A more formal description of the VERIFY task is provided by the Verification Algorithm which follows.

Algorithm for Verification

This algorithm verifies a formula by processing the atoms of each subformula.

Let SUBFORM_LIST be a pointer to the list of subformulae of a formula. Let SUBFORM traverse this list. Let ATOM be the atom currently being examined. ORDER(X) is a function with one argument, a subformula pointer. It returns a pointer to a linked list of the atoms of the subformula in the order in which they should be examined. It does this by applying the heuristics discussed above. EXECUTABLE(X) is a predicate which is true if the atom X contains an executable predicate and false if it contains a COPE relation. APPLY(X) is a function which executes the predicate in the atom X, if it is sufficiently instantiated, and returns the value. If the predicate cannot be executed yet, the value returned is null. MARK(X,Y) is a function which marks the atom X with the value Y. As a side effect, when an atom is assigned a value of true, MARK constructs a new node of the proof tree for that atom. RETRIEVE(X) is a function which invokes the LOCAL DATA task to retrieve tuples matching atom X from the data base. If the LOCAL DATA task retrieves anything, RETRIEVE, as a side effect, instantiates the variables of atom X. RETRIEVE also manages the data which is part of a context. This user specific data base has the same structure as the local data base, so that the same accessing algorithms apply, but the data is only available to one user. TEST(X) is a predicate which is true if the subformula represented by the atom list X is true, false if the subformula is false, and null if the truth value of the subformula cannot be determined.

C1: [Initialize.]

SUBFORM := NEXT(SUBFORM_LIST)

C2: [Get next subformula.]

IF SUBFORM = NULL THEN TERMINATE
SBF := SUBFORM

[The algorithm terminates when all subformulae of a formula have been tasked.]

C3: [Order atoms.]

SBF := ORDER(SBF)
ATOM := SBF
GO TO C5

[A linked list of the atoms of the current subformula is created by ORDER and the pointer to it is stored in SBF and ATOM.]

C4: [Get next atom.]

```
ATOM := NEXT(ATOM)
IF ATOM = NULL THEN
    Notify the M/R task that more information is needed.
    Suspend the VERIFY task.
```

[When all atoms of a subformula have been tested and a truth value has not been determined, suspend processing of the context until the user supplies more information.]

C5: [Type 1 atoms.]

```
IF EXECUTABLE(ATOM) = TRUE THEN
    VAL := APPLY(ATOM)
    IF VAL = NULL THEN GO TO C4
    ELSE
        MARK(ATOM,VAL)
        GO TO C7
```

[If the atom is type 1 (executable), attempt to execute the predicate. If a value of null is returned, the atom is not sufficiently instantiated to be executed, so try the next atom. Otherwise, mark the atom with the result of the operation.]

C6: [Type 2 atoms.]

```
MARK(ATOM,RETRIEVE(ATOM))
```

[RETRIEVE invokes the LOCAL DATA task to retrieve the data which matches the atom pattern. If matches are found, RETRIEVE instantiates the variables of the atom. The atom is marked true if matches are found and false if there are no matches.]

C7: [Test subformula.]

```
VAL = TEST(SBF)
IF VAL = TRUE THEN
    SUBFORM := NEXT(SUBFORM)
    GO TO C2
IF VAL = FALSE THEN
    Note the constraint rule violation condition
    Notify the M/R task of the violation
    Suspend the VERIFY task
ELSE
    GO TO C3
```

[If the value of the subformula is true, terminate processing of its atoms and begin the next subformula. If the value is false, a

constraint violation has been detected, so report the problem to the user via the M/R task and suspend the context. If a truth value cannot be determined, reorder the atom list if necessary and traverse the list again.]

As each subformula is completed, the algorithm is repeated with the atoms from the next subformula. When a formula is completed, VERIFY selects the next rule from the list ordered by the search strategist.

4. Backtrack Algorithm

The following algorithm specifies how BACKTRACK examines and modifies a context when invoked by either the LOCAL DATA task or the M/R task:

Algorithm for Backtracking

This algorithm uses the access list of each changed datum to determine what parts of a context are affected, then reverifies those parts, pruning the proof tree as necessary.

CHANGE_LIST is a list of the data items changed by a user. Each entry in the list has three fields, the old item (OLD), the new item (NEW), and a link to the next entry (NEXT). CHANGE traverses CHANGE_LIST. OLD and NEW each point to data nodes which have two fields, the actual data fact (DATUM) and a pointer to a list of users who are accessing the data (ACCESS). The ACCESS list and INSTANCE_LIST were described in section 5.5. They are traversed by ACC and IPL, respectively. SINGLE_USER is a flag that is 'TRUE' if BACKTRACK was called because of a user changing his context and thus affecting no other users. Since he is the only user of the data, there is exactly one entry on ACCESS. SINGLE_USER is 'FALSE' if BACKTRACK was called due to a user's inputs being passed on to the DBMS. MASTER_USER is a flag which indicates the user ID of the user who caused the data base update. RECOPY(X) is a function which deletes nodes from an ACCESS list X that have blank INSTANCE fields and returns a pointer to X. MATCH(X,Y) is a predicate that is true if datum Y satisfies atom X. As a side effect, the variable instantiations in the atom X are changed when necessary and the affected variables are marked. PASSON(X) is a predicate that performs reverification by examining things that have changed due to changes in the instantiation of variables. It is 'TRUE' if the reverification is successful and 'FALSE' if a violation is detected. X is a pointer to the rest of the proof tree that must be tested.

D1: [Initialize; get first change.]

CHANGE := CHANGE_LIST
GO TO D3

D2: [Check for end of changes; get next change.]

CHANGE := NEXT(CHANGE)
IF CHANGE = NULL THEN TERMINATE

D3: [Get first user of this data.]

ACC := ACCESS(OLD(CHANGE))
GO TO D5

D4: [Get next user; check for last user.]

```
ACC := NEXT(ACC)
IF ACC = NULL THEN
    ACCESS(NEW(CHANGE)) := RECOPY(ACCESS(OLD(CHANGE)))
    GO TO D2
```

[If done with every user of this data, call RECOPY to delete nodes in the ACCESS list representing instances where the data is no longer being accessed. This revised ACCESS list becomes the ACCESS list of the new data.]

D5: [Get first instance.]

```
IF SINGLE_USER = FALSE THEN
    IF MASTER_USER = USER(ACC) THEN GO TO D4
IPL := INSTANCE_LIST(ACC)
```

[If BACKTRACK was invoked due to a single user updating his own context only, then there is only that user on the access list; perform backtracking for him. If the task was invoked due to a user's inputs being passed on to the DBMS, then perform the algorithm for all users except the one who effected the update (MASTER_USER). Point to the first instance for this user.]

D6: [Check each instance.]

```
IF IPL = NULL THEN GO TO D4
INST := INSTANCE(IPL)
```

[Point to the data entry for the next instance until all instances have been considered.]

```
IF MATCH(ATOM(HEAD(INST)), DATUM(NEW(CHANGE))) = TRUE THEN
    DATUM(INST) := DATUM(NEW(CHANGE))
```

[If MATCH determines that the new data item satisfies the atom pointed to by this data entry, then substitute the new data for the old in the DATA_LIST for this atom.]

```
IF PASSON(NEXT(HEAD(INST))) = TRUE THEN
    IPL := NEXT(IPL)
    GO TO D6
ELSE
    GO TO D7
```

[PASSON attempts to reverify the remainder of the tree which may be dependent on the changed item. If it is reverified successfully, then point to the next instance and repeat these tests. Otherwise, a violation has been detected.]

```
ELSE
    INSTANCE(IPL) := NULL
```

[If the new data cannot replace the old, a constraint violation has been detected. Set the INSTANCE field of the INSTANCE_LIST entry to null so it will be deleted.]

D7: [Constraint violation.]

Note the constraint rule violation condition
Notify the M/R task of the violation
Suspend the BACKTRACK task

